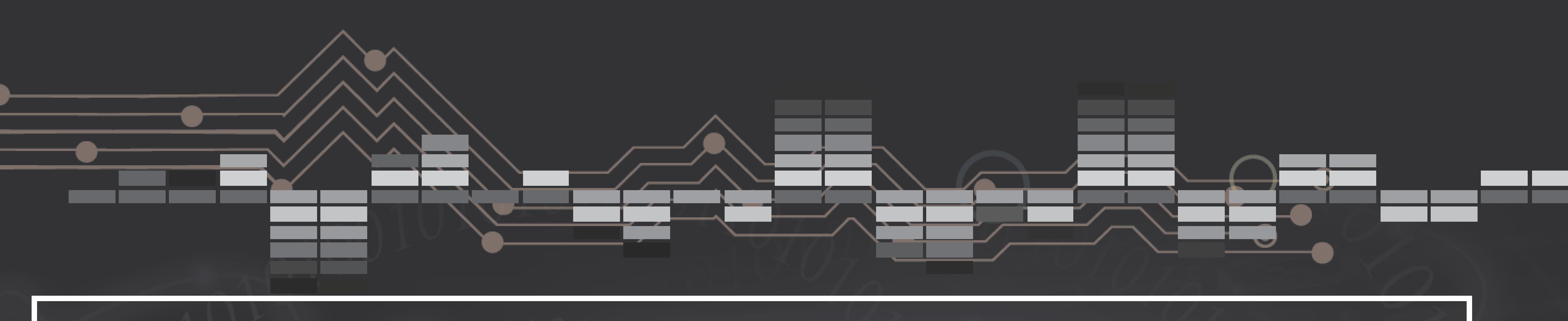
微软专家级
工程师力作



Web前端测试与集成

Jasmine/Selenium/Protractor/Jenkins的最佳实践

金鑫　武帅　编著



以实际项目为范例，内容丰富，示例与图解清晰，全方位介绍Web前端开发测试中的单元测试、自动化测试以及集成测试

清華大学出版社

Web 前端测试与集成：

Jasmine/Selenium/Protractor/Jenkins 的最佳实践

金鑫　武帅　编著

清華大學出版社
北　京

内 容 简 介

近些年，为了适应移动设备和云端服务，Web 前端涌现出很多新框架和新技术，这为高质量的软件开发带来了前所未有的挑战。本书详细介绍了 Web 前端开发与测试的理论，以及基于 Jasmine、Selenium、Protractor 和 Jenkins 如何进行全生命周期的测试与集成。全书共分四个部分。第一部分为基础篇，总览了前端开发测试中的挑战与测试转型，介绍了测试基础环境的搭建；第二部分为单元测试篇，深入介绍了如何基于 Jasmine 单元测试框架和 gulp、Karma 等构建、执行工具对前端 JavaScript 代码进行单元测试，以及 AngularJS 单元测试的最佳实践和代码覆盖率等；第三部分为自动化测试篇，基于 Protractor 介绍了在 Node.js 环境下通过 Selenium WebDriver 全面覆盖各个主流浏览器，进行自动化测试的最佳实践，包括页面对象模型、性能测试和分布式测试等；第四部分为集成篇，阐述了基于持续集成以实现更快、更可靠的软件交付，展示了如何通过 Jenkins 与 TFS、VSTS 和 GitHub 的集成，实现 Web 应用的持续测试。

本书兼顾了当前 Web 应用的各项前沿技术，采用其最新版本，内容丰富，示例与图解清晰易懂，适合所有 Web 开发人员、测试人员和项目经理做学习、参考之用。

图书在版编目(CIP)数据

Web前端测试与集成：Jasmine/Selenium/Protractor/Jenkins的最佳实践 / 金鑫，武帅编著. —北京：清华大学出版社，2017

ISBN 978-7-302-47275-9

Ⅰ. ①W… Ⅱ. ①金… ②武… Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2017）第 123323 号

责任编辑： 袁金敏
封面设计： 刘新新
责任校对： 徐俊伟
责任印制： 王静怡

出版发行： 清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者： 三河市金元印装有限公司
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　　张：** 24.75　　**字　　数：** 497 千字
版　　次： 2017 年 9 月第 1 版　　**印　　次：** 2017 年 9 月第 1 次印刷
印　　数： 1～3000
定　　价： 69.00 元

产品编号：073070-01

前　言

笔者多年来一直在微软公司从事与Web开发相关的技术工作，帮助客户维护和处理各种Web应用的突发事件，期间走访了大量客户，为他们提供解决方案或指导意见。我们深深感受到近些年来Web技术的快速发展对Web开发人员、测试人员带来的诸多挑战：

- Web开发是一个开放的、不断演进的、高速迭代的领域。即便是过去一直被诟病为封闭的微软公司也开始拥抱开源世界，提供.NET Core、TypeScript等多项开源和跨平台技术。这对于过去长期在单一厂商平台上进行项目开发的技术人员而言，就更需要积极主动地学习新技术，接受新挑战，以适应变化，满足业务需求。
- 基于JavaScript的前端应用规模越来越大，功能越来越复杂，前端测试已经成为保证产品质量的关键因素。同时，由于Web开发存在测试周期短、更新频繁的特点，传统测试人员需要具备一定的开发能力才能充分利用自动化测试工具来提高测试效率。
- 随着敏捷软件开发方法和DevOps的流行，测试和开发环节之间的界限逐渐变得模糊。传统开发人员需要了解一定的测试方法并具有相应的思维方式，才能设计出良好的测试用例。由于测试和开发环节的融合，无论是开发人员还是测试人员都需要不断提高自身的能力和价值。

本书是笔者在开发测试领域中的实践与经验的总结，希望读者通过对本书的学习，能够掌握Web前端测试的各种技巧，提升自己的能力，迎接新技术的挑战。

本书内容

全书共分四个部分，前两部分为金鑫编写，后两部分为武帅编写。

- 第一部分为基础篇（第1～2章），总览了前端开发测试中的挑战与进行测试转型的方法，以及基于Node.js搭建测试开发基础环境的步骤。
- 第二部分为单元测试篇（第3～7章），基于单元测试理论深入剖析了Jasmine测试框架的结构与各种使用范式，内容覆盖了所有主流单元测试的技巧。然后，结合

gulp、Karma等构建、执行工具对单元测试进行自动化处理。最后以实战的形式演示了AngularJS单元测试最佳实践以及Istanbul代码覆盖率的应用等内容。

- 第三部分为自动化测试篇（第8～14章），由浅入深地介绍了Selenium各个组件的功能特点和WebDriver在自动化测试中的使用技巧。进而基于Protractor深入介绍了在Node.js环境下，通过JavaScript代码结合WebDriver进行自动化测试，并全面覆盖Chrome、Firefox、IE和Edge等主流浏览器的最佳实践，内容包括页面对象模型、性能测试、数据驱动测试和分布式测试等。
- 第四部分为集成篇（第15～16章），阐述了基于持续集成技术来实现更快、更可靠的软件交付方法，比较了当前主流CI系统的特点，展示了通过Jenkins与TFS、VSTS和GitHub集成来实现Web应用持续测试的方法。

本书适合对象

本书适合所有Web开发人员、测试人员和项目经理做学习、参考之用。本书涉及的示例代码，读者可从网址https://github.com/FrontEndTesting/webtesting-book-demo处下载，供对照学习。

致谢

首先，感谢家人对我们利用业余时间编写本书的理解，在漫长的编写过程中始终给予关爱和支持。其次，感谢张量、毛蔚、徐春林和顾洁等微软公司同事的大力鼓励与支持，本书的成书与他们密不可分。

由于笔者学识有限，时间仓促，书中难免出现错误或疏漏，恳请广大读者不吝指正。如果您有什么宝贵意见，请发送邮件至jin_xin2000@outlook.com，我们将不胜感激。

目　录

基础篇

单元测试篇

自动化测试篇

基础篇

第1章 前端开发测试总览

软件的本质是靠良好的质量吸引客户。近些年，随着新技术新模式的出现，企业在交付高质量软件和服务方面，面临着越来越大的压力。正因为如此，对软件产品的质量控制已成为企业生存和发展的核心，几乎所有的企业，在软件开发生命周期里都会持续投入测试力量。

同时，Web开发技术为了适应移动设备和云端服务的应用趋势，发生了很大的变化和演进，各种新标准、新框架、新工具、新理念如雨后春笋般涌现。如何在测试中从容应对各个框架的新特性，全面覆盖各个主流浏览器，为Web测试带来了前所未有的挑战，也迎来了测试自动化转型的契机。

本章将介绍：

- Web技术的发展和挑战
- 传统开发流程的局限性
- 传统手工测试的局限性
- 开发模式的转型

1.1 Web技术的发展和挑战

Web应用通常分为前端和后端，前端主要在浏览器里显示和采集信息，后端主要处理数据和业务逻辑。传统的Web应用一直偏重后端开发。当用户通过浏览器访问一个网址，这个网址可能是一个存储在服务器上的静态HTML文件，由服务器读取后直接返回。更多的情况是这个网址对应的是一个模板，服务器在用户请求时根据业务逻辑和模板动态拼接成HTML格式的字符串。Web开发人员采用各种后端动态页面技术比如CGI、PHP

和ASP等生成这样的字符串。换句话说，所谓的“前端”是由后端服务器动态输出而成的HTML网页。

在这样的传统Web应用中，前端只是充当一个展示层，服务调用、网页跳转流程等都需要由后端服务器处理。每当网页功能有所变化时（即使网页内容只有细微变动），浏览器都需要重新发起一个HTTP请求到服务器，然后由后端服务器重新生成整个页面如图1-1所示。这种方式不但增加服务器的负担，降低响应速度，而且浏览器重复下载整个网页会浪费网络带宽，对于移动应用很不经济。

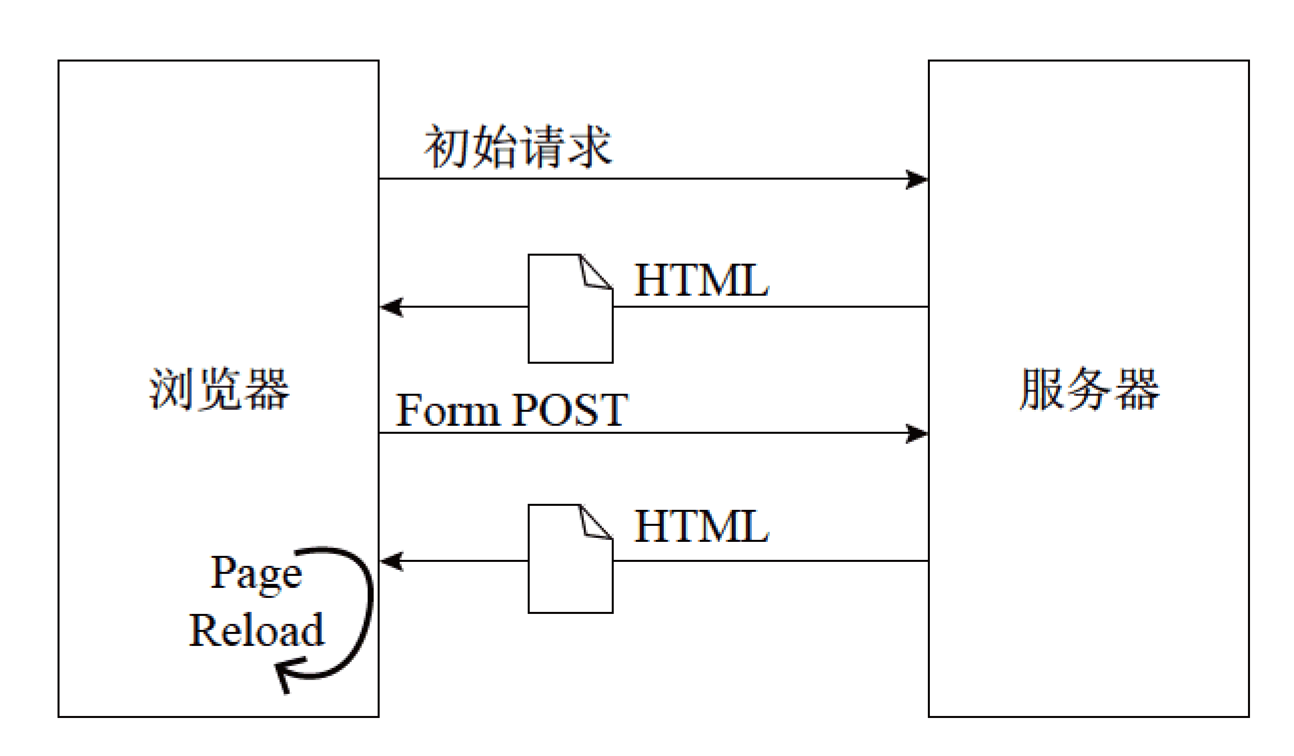


图1-1　传统网页生命周期

而在现代Web应用中，当网页的部分内容需要更新时，运行在浏览器内的JavaScript代码通常会向服务器发送Ajax请求，而服务器端只需输出必要的数据，不需要重新构造整个HTML页面，如图1-2所示。Web前端应用接收到来自服务器的数据后，只需重绘界面上需要变化的部分。这种方式提高了应用的响应速度，改善了用户的使用体验。同时，因为很多用户交互的处理工作可以在浏览器内完成，无需向服务器发送HTTP请求，服务器和浏览器之间交换的数据量大幅减少，所以服务器负荷降低，响应速度也更快了。Web应用渐渐开始拥有和桌面应用一样的响应速度和使用体验。

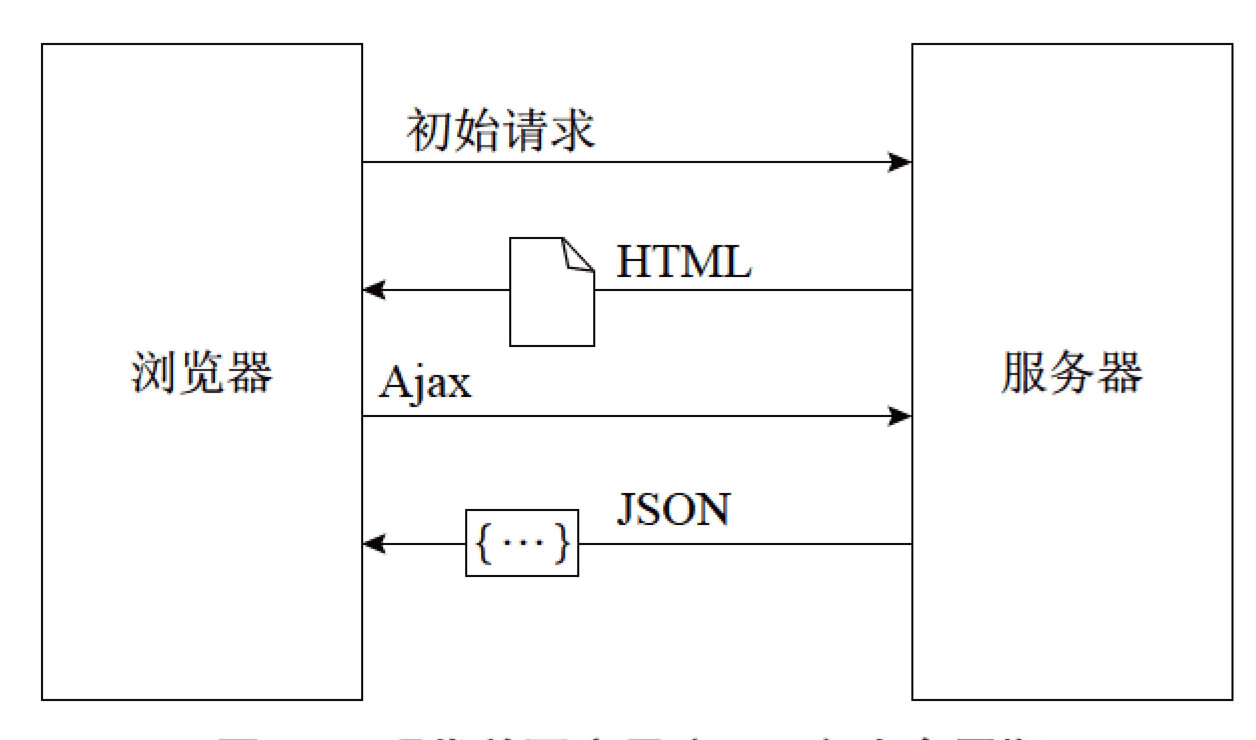


图1-2　现代单页应用（SPA）生命周期

除了网页内容的更新，Web应用一部分业务逻辑的处理也从后端服务器慢慢移到前

端，Web前端开发变得越来越重要。而这一切都离不开JavaScript。

JavaScript是1995年由Netscape工程师Brendan Eich花了10天的时间创造出来的，其最初目的是为了满足用户浏览网页时产生的交互需求。可以说，JavaScript是为开发Web应用而诞生的。

自其诞生起的相当长一段时间内，这门在浏览器内执行一些简单的校验和互动的脚本语言常被称之为“玩具语言”，堪称世界上被人误解最深的编程语言[①]。在简洁的外表下，JavaScript其实有着强大的语言特性。它是一种面向对象的动态语言，包含类型、运算符、标准内置对象和方法。其语法来源于 Java和 C，所以这两种语言的许多语法特性同样适用于JavaScript。

随着Web应用的普及，JavaScript自身也在不断演化以适应更复杂的编程需求，目前JavaScript已经成为Web前端的实际标准，也是最热门的编程语言之一。各种JavaScript前端框架，包括Ember、AngularJS、React和Vue等不断涌现。

正因为基于JavaScript前端应用越来越重要，功能越来越复杂，对前端开发测试带来了极大的挑战。如何保证Web前端应用的正确性和可靠性成为了各大企业关注的问题。

- 在市场需求的推动下，Web前端应用规模不断扩大，一个应用可能有成千上万行JavaScript代码。这些代码用于执行各种复杂的功能，为用户提供不同的交互体验。为了确保它们能实现预期的功能，因此Web前端应用测试的工作量也需要相应增加。
- 相对于传统的桌面应用，用户只要重新刷新浏览器就可以获得最新版本的网页，所以Web前端应用具有天生的快速迭代特征，可以频繁地更新和发布。而激烈的商业竞争也使得Web前端应用开发周期缩短，对应的测试周期也非常短。
- 除了JavaScript代码，Web前端应用还有各种超链接、表单以及图片等信息，需要保证在不同的操作系统和浏览器上可以正确显示网页的内容。

1.2 传统开发流程的局限性

在过去二十年或更长的时间中，传统的、非敏捷的瀑布式软件开发模式通常依赖于一

① Douglas Crockford. JavaScript: The World's Most Misunderstood Programming Language[OL]. [2016]. http://javascript.crockford.com/javascript.html.

个严格的模式化开发流程。通常认为瀑布模型[①]是Winston W. Royce在1970年提出的软件开发模型（虽然他并没有使用瀑布waterfall这个单词）。这种方法源自传统工业生产，严格遵循预先计划的需求、分析、设计、编码、测试和部署的步骤顺序进行，每个步骤都有严格分工，由不同的技术人员分别执行。执行步骤的成果作为衡量进度的途径，例如需求规格、设计文档、测试计划和代码审阅等。但随着各种新兴技术的蓬勃发展，特别是在产品快速迭代的需求背景下，这种传统开发流程的局限性日益明显：

1. 自由度低，缺乏灵活性

传统模式在项目早期即作出承诺，基于稳定的目标进行阶段性的开发，这种方式自由度低，应对突发情况时缺乏灵活性。众所周知，需求会随着时间而变化，经常出现开发人员努力在设计前完成文档，在编写代码前完成设计，最后却因为需求有变化而不得不完全推倒重来的情况，不仅大量工作被浪费，甚至导致对后期需求的变化难以应变，代价高昂。

2. 缺陷发现晚，无法及时反馈

传统流程一般在开发阶段接近尾声时才开始测试。虽然在这个阶段进行测试相对容易，但是一些在早期的单元测试中可以轻易发现的缺陷可能要到最后阶段才会发现，增加了被遗漏的风险。

由于缺陷发现得很晚，如果要解决问题，则有可能导致错过发布的最后期限，再加上手工测试效率低下，任何一次代码变更或缺陷修复对产品的影响都无法迅速反馈给开发人员。随着发现的缺陷越来越多，开发人员只会对变更没有信心，失去持续完善的动力。

3. 协同合作缺失，容易引起团队冲突

传统流程中每个步骤都有严格分工，不同阶段的技术人员与上下游的工种往往沟通较少，甚至对产品本身的理解也存在差异。例如，开发人员和测试人员在思维和工作方式上的不同，使得他们对软件需求和测试场景的表述发生歧义，从而引起双方的沟通问题。而这些理解上的差异如果直到测试阶段才被发现的话，则实在太晚，此时双方的冲突与指责对解决问题没有任何帮助。

4. 产品质量无法保证

传统流程中常见的一个场景是在项目后期将要交付的阶段，技术人员仓促地从开发环境构建转为脚本配置，而长期在开发环境内依靠手工管理，构建脚本的时候就可能会遇到各种各样的问题，例如接口没有定义、配置文件丢失、组件不工作等。为了解决这些问

---

① Wikipedia. Waterfall model[OL]. [2016]. https://en.wikipedia.org/wiki/Waterfall_model.

题，技术人员不得不把主要精力放在软件构建上，结果时间成本越来越高，产品质量无法保证，甚至出现产品无法按时交付的情况。

1.3 传统手工测试的局限性

软件测试是在规定的条件下对程序进行操作，以发现程序中的错误，衡量软件质量，并对其是否能满足设计要求进行评估的过程。软件测试的目的是希望以最小的代价尽可能多地找出软件中潜在的错误和缺陷。

首先，测试人员会针对开发人员开发的功能写出测试用例，例如表单应该填入的数据，页面单击顺序，以及最后页面期待的显示效果。然后，测试人员会按照用例一步步进行手工检验，如果发现页面行为异常，例如无法打开页面或生成的数据不正确，则会在企业缺陷管理系统中提交缺陷记录，供开发人员进行修正。在开发过程中，如果有新版本编译出来，测试人员需要根据测试用例重新测试，确认是否有新缺陷，或者老缺陷是否已经得到了修正。

长久以来，这种传统的手工测试模式在各大公司广泛应用，并已被证明其能够行之有效地保证产品质量。但伴随着互联网技术的发展，这种传统的测试模式已经显示出越来越多的瓶颈。

1. 重复性工作，测试质量低

现在的互联网产品开发讲究的是短平快，小步快走，短则两三天，长则一个星期就会发布新版本。在这短短的时间里，测试人员需要把新版本部署到测试环境，更新数据库，然后对所有测试用例进行手工验证。这个过程时间紧迫，工作量大，而且具有很高的机械性和重复性。当测试人员长期工作在重复性的验证事务上时，往往会因为思维惯性而忽略新出现的问题，最后导致不仅测试人员自身缺乏工作热情，而且测试质量更难以保证。

2. 测试效率低

手工测试天生就决定了它的执行效率很低。测试人员需要根据测试用例逐行逐句阅读，然后在页面上一步步填写表单，再单击按钮提交。这是一个非常烦琐的过程。而遇到复杂的业务流程更是涉及方方面面，作者甚至见过一个多小时都无法完成的测试案例。到了开发后期，可能每天或者每两天就要发布一个版本进行测试。如果一个软件系统的功能点有几千甚至上万个，手工测试将特别耗时和烦琐，不仅消耗了大量的人力，还可能影响

到产品的如期发布。

3. 无法保证覆盖代码全路径

是否有良好的测试覆盖是考核测试成熟度的重要指标，其核心思想是对相同的业务逻辑提供多组甚至几十组输入，全面覆盖到业务中的绝大多数路径，重点考察软件的边界行为。比如某个页面输入框的字符个数在开发中被限制为256个字符，但测试人员很可能漏掉这样的极端输入情况。由于手工测试效率很低，不要说进行几十组数据的测试，就是几组可能都难以实施。另外，有些软件缺陷需要在大量数据或者大量并发用户的情况下才会暴露，很难通过手工测试保证代码的全路径覆盖。

4. 无法有效兼顾多浏览器、多平台

Web前端的测试环境复杂，兼容性要求高，特别是要同时兼顾多种操作系统，包括Windows、Mac OS和Linux，以及不同的浏览器，包括IE、Edge、Chrome、Firefox等，如果还考虑各个操作系统和浏览器的不同版本，排列组合之后将会是个通过手工测试无法企及的数字。很难想象有哪个公司能够持续投入巨大的人力成本完成如此庞大的手工测试工程。

1.4　开发模式的转型

针对传统开发流程和手工测试的局限性，各大企业迫切需要对开发模式进行转型，以应对现代Web应用开发周期短，更新频繁等挑战，同时做到尽早识别测试风险，通过合理的应对策略保证产品质量。

1.4.1　敏捷软件开发

敏捷软件开发（Agile Software Development）是一类已经引起广泛关注的软件开发方法，是为应对需求快速变化而发展出的软件开发方法。有多种敏捷开发方法，例如极限编程（Extreme Programming）、精益开发（Lean Software Development）、特征驱动开发（Feature-Driven Development）等，它们有以下共同的特征，如图1-3所示：

- 迭代式开发。整个开发过程被分为几个迭代周期，每个迭代周期持续的时间一般较短，通常为1～4周。

- 增量交付。产品是在每个迭代周期结束时被逐步交付使用的，每次交付的都是可以被部署、能给用户带来即时效益和价值的产品。
- 及时反馈。敏捷软件开发主张用户能够全程参与到整个开发过程中。这使需求变化和用户反馈能被动态管理并及时集成到产品中。
- 关注软件质量。在开发的整个周期中都关注产品的质量。开发过程中使用的各种工具和方法，例如持续集成、测试自动化、测试驱动开发等都为敏捷项目的整个开发周期提供了可靠的质量保证。

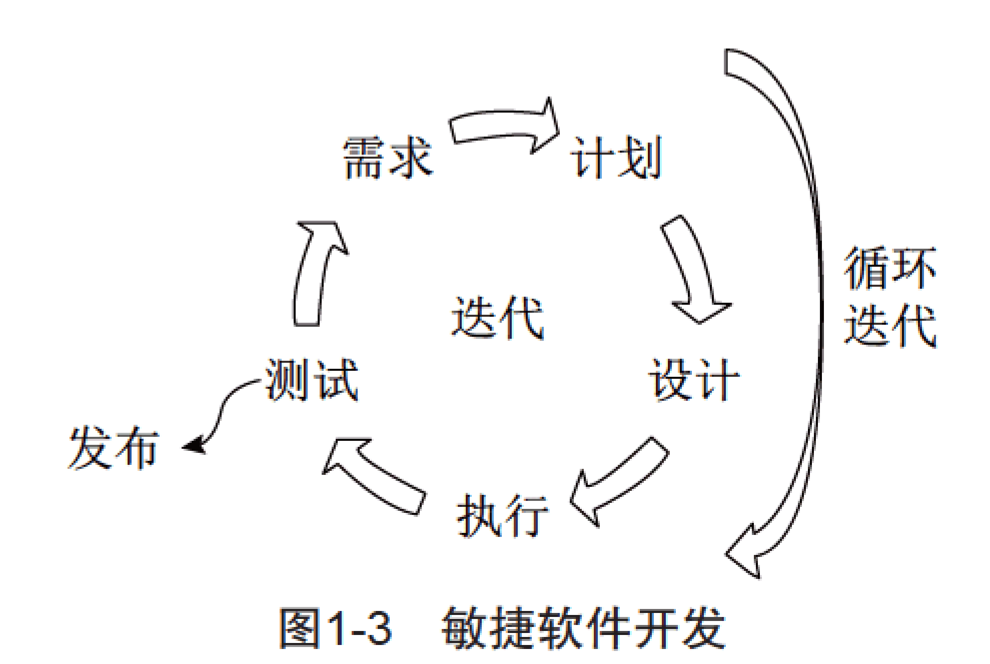


图1-3　敏捷软件开发

因为敏捷软件开发拥有更强的灵活性、更短的开发周期、持续反馈等优点，所以敏捷开发被越来越多的软件开发企业和团队所接受。

1. 更强的灵活性

相对于传统的瀑布模型，敏捷开发尝试以更加灵活的方式让每个开发阶段都并行发生，更强调开发周期内开发团队与客户、开发团队内各个角色之间的紧密协作和有效交流，以便更早发现问题，从而降低改正问题的成本和提高项目成功的几率。

2. 更短的开发周期

敏捷开发是将一个大项目分为多个相互联系，但也可独立运行的小项目，并分别予以完成。这种模式强调的是尽早将可用的功能交付使用，并在整个项目周期中持续改善和增强。更重要的是，在每个迭代周期中，功能特性被开发和测试，所有发现的问题都被及时修正。这样，开发人员和测试人员之间的时间鸿沟就消失了，因为他们始终在相同的迭代周期中协作。

3. 持续反馈

敏捷开发短而多的迭代周期为功能调整提供了可能性。用户能够全程参与到整个开发过程中，敏捷团队几乎可以在任何时间满足用户不断变化的需求。

4. 测试和开发技能的融合

在敏捷软件开发中，测试和开发之间的界限变得模糊。一方面，当敏捷团队配备相对

独立的测试人员时，测试人员往往需要有一定的开发能力，才能和开发人员紧密配合完成测试，满足项目进度的要求。另外一方面，当测试角色由开发人员兼任的时候，开发人员需要培养自己良好的测试技能，包括测试用例的设计开发以及执行和结果分析能力。

1.4.2　全流程测试

作为保证软件质量手段之一的测试，不应该仅仅局限于软件开发中的某个阶段，它应该贯穿于整个软件开发的全过程。测试开始的时间越早（test early），测试执行越频繁（test often），就可以越早暴露和发现软件系统存在的质量风险。根据测试在软件开发过程中所处的阶段和作用，可分为单元测试、集成测试和端到端测试等。

1. 单元测试（Unit Test）

软件开发过程中，最基本的测试就是单元测试。这是针对程序单元（软件设计的最小单位）来进行正确性检验的测试工作。程序单元是应用的最小可测试部件。在过程化编程中，一个单元就是单个程序、函数、过程等；对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类或者派生类（子类）中的方法。在企业的质量控制体系中，单元测试由开发部门在软件提交测试部门前完成。

单元测试的目标是打破程序单元间的依赖关系，隔离单元并证明这些单个单元是正确的，所以单元测试应该无依赖和隔离。通常在单元测试中，把系统的依赖组件提取出来，用测试替身（Test Double）取而代之，把单元测试把注意力集中放在测试“单元”的逻辑上而不是和第三方系统的交互上，如图1-4所示。常见的依赖组件有网络、数据库、第三方类库和文件系统等。

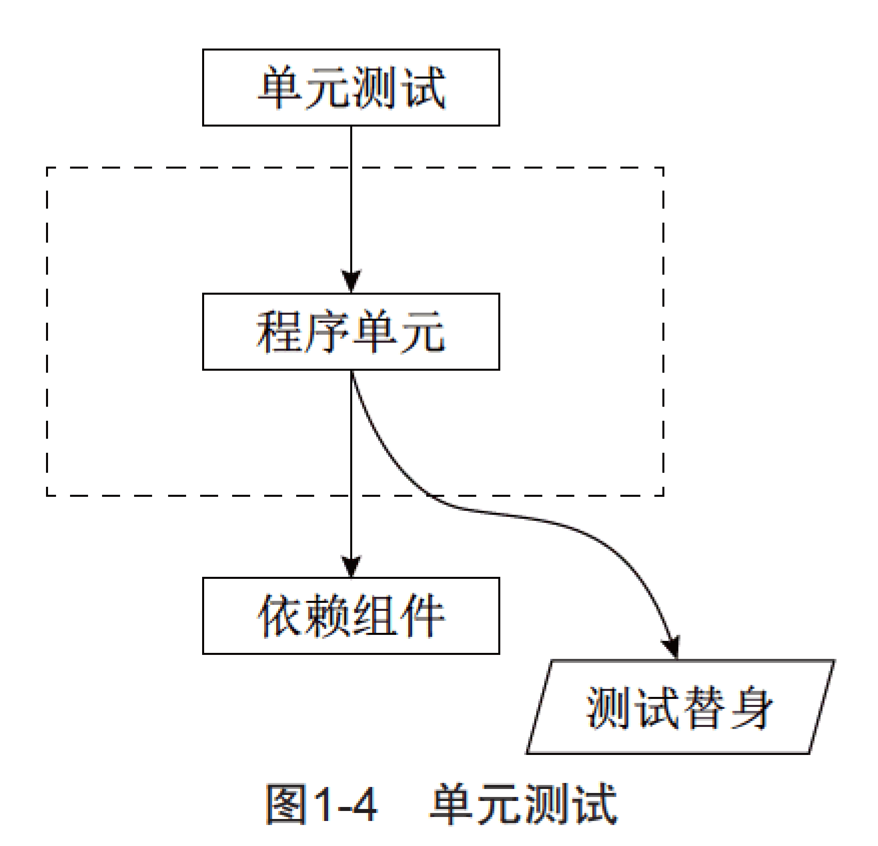


图1-4　单元测试

2. 集成测试（Integration Test）

即使一个程序单元在隔离状态下运作良好，也并不能确定它们放在一起能正常工作。

集成测试是取出应用程序里可以独立运行的组件，通常是一些单元的集合，来测试这些单元作为一个整体的表现，以验证它们能否协调一致地运作（如图1-5）。集成测试一般用于单元测试之后，端到端测试之前。

例如一个常见的集成测试场景是使用数据组件对数据库进行操作的测试。测试人员需要安装并配置好数据库，然后在数据库里插入预先准备好的数据，再执行需要测试的组件，运行完毕后检验数据库里的数据。在这个测试场景中，被测的单元依赖数据库访问模块（例如Microsoft Entity Framework），和一台真实的数据库（外部系统），所以它不是一个单元测试，但是它也没有模拟一个完整的用户真实场景，所以它也不是一个端到端测试。

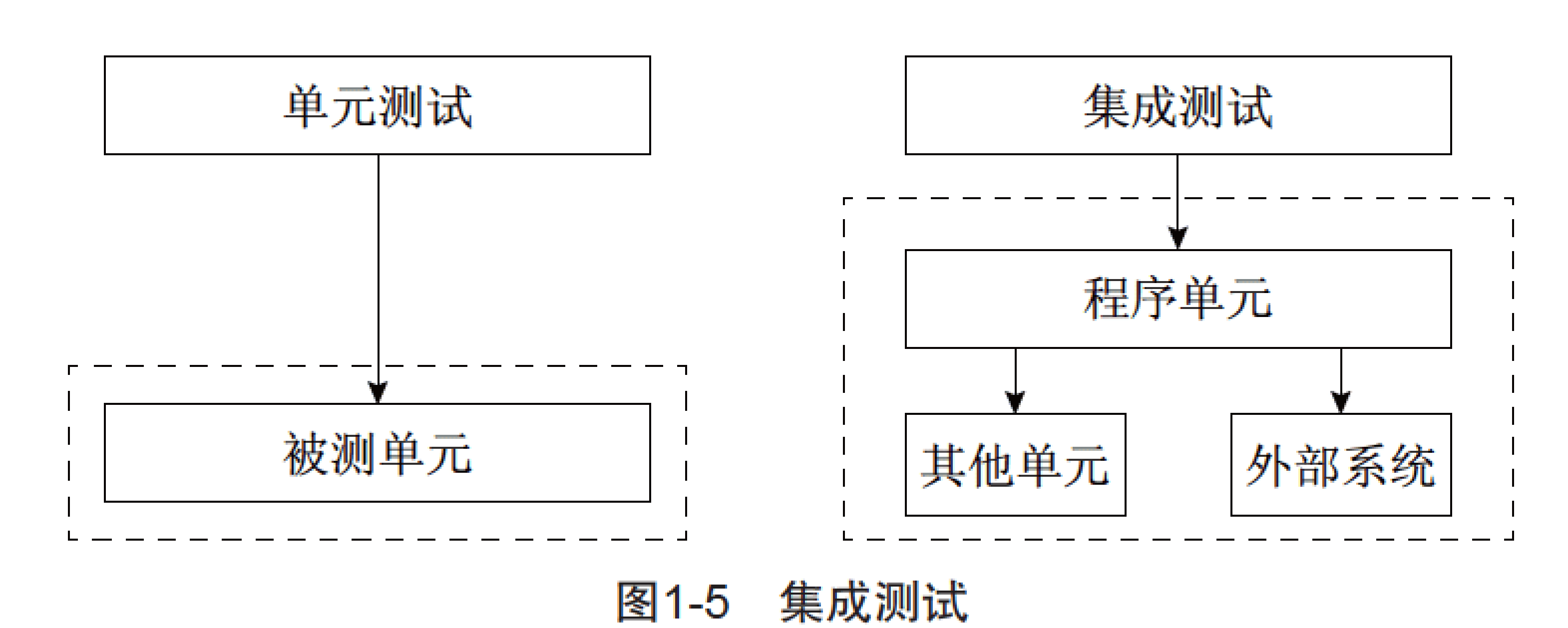


图1-5 集成测试

3. 端到端测试（End-to-End Test）

端到端测试（缩写为E2E Test）是把产品或服务当作一个整体进行验证。典型的做法是模拟真实的用户场景，通过与系统的需求定义作比较，来发现产品与需求定义不符合或存在矛盾的地方，其最终目的为了发布产品。例如在Web应用程序中，测试人员会启动服务器，打开浏览器，访问被测网页，并操作网页上需要测试的功能，检查浏览器中发生的特定的事件，以确保被测功能可以正常运行。

端到端测试通常由测试部门完成，一般有以下特性：

- 需要搭建专门的测试环境模拟真实的用户场景，成本较高。
- 测试用例复杂，运行时间长。
- 一旦测试发现问题，由于涉及的模块比较多，定位问题难度较高。

端到端测试可以手工完成，也可以编写测试框架和测试代码自动执行。在Web前端应用中，端到端测试通常从用户界面开始，核实用户与应用之间的交互，确保用户界面向用户提供了适当的访问测试对象功能的操作，同时还要确保内部的对象符合预期要求。如果进行手工测试的话，效率低下，无法满足快速迭代的Web前端应用的测试需求，所以迫切需要将Web前端应用的端到端测试自动化。本书第三篇介绍的自动化测试指的就是自动化的端到端测试。

1.4.3　让测试自动化

相对于手工测试，测试自动化是把以人为驱动的测试行为转化为机器执行的一种过程。需要避免的误区是，测试自动化并不是要彻底摆脱测试人员，而是一种由人设计机器行为，让机器驱动测试的新模式。

实施测试自动化后，执行者是机器，它可以24小时不停地运行测试代码，充分利用硬件资源，提高测试效率。对于一些手工完成困难或不可能进行的测试，例如测试大量用户同时在线的情况，测试自动化也可以模拟这些用户，提高测试用例的广度。测试自动化对Web前端应用的回归测试效果也非常明显。Web前端应用更新频繁，因为测试自动化的测试用例可以重复使用，保证了测试环境和测试路径的一致性，这不仅可以缩短回归测试的时间，而且很容易发现代码修改引起的回归缺陷。

1.4.4　持续集成

测试自动化是进行快速迭代开发的关键一步。然而，如果测试用例执行失败了，是否有一个清晰的工作流程以优先级排序形式标注软件缺陷，反映与商业风险的关联关系，以及列出哪些是急需解决的问题？同时，随着软件开发复杂度的不断提高，团队成员间如何更好地协同工作以确保软件质量，能否通过流程管理解决软件开发的上下游协作？这些已经成为开发过程中不可回避的问题。软件开发急需一种自我管理、自我适应，让开发自动化起来的新模式。

持续集成（Continuous Integration）是一个频繁持续的在团队内进行业务集成，自我反馈完善的软件开发实践。根据Martin Fowler的观点[①]，持续集成要求团队成员经常集成他们的工作，每个人至少每天集成一次。持续集成通过自动化构建，把包括编译、部署、测试、审计和反馈的一组流程用一体化方案驱动起来，整个流程不需要任何用户的人工干预。

持续集成的好处有：

- 可以及早发现缺陷。持续集成要求每天多次进行集成并执行测试和审查，这可以确保新增代码不会破坏之前的运作。即使出现了回归缺陷，开发人员也可以迅速获得通知，及时修复缺陷。

---

① Martin Fowler. Continuous Integration[OL]. 2006. http://www.martinfowler.com/articles/continuousIntegration.html.

- 通过构建自动化过程，减少开发测试人员的重复劳动。
- 团队成员在任何时间点上提交的代码都可以进行集成，这使得开发团队能随时发布可部署的软件。
- 持续集成良好的架构可以有效实现分布式团队的协作沟通，让团队成员任何时候都能了解产品的状态，实时地知道当前已经完成了什么功能，还有什么缺陷需要修复。

1.4.5 DevOps

在传统的软件开发中，开发、测试和运维都有独立运行的部门。虽然敏捷软件开发模糊了需求、架构、开发、测试之间的界限，但是各部门间仍然存在着信息“鸿沟”。例如运维人员更关注产品的日常运作、可靠性和安全性，而开发人员通常把主要精力放在新功能的开发上。

现代Web应用和移动应用需要频繁而持续的发布。软件产品会被部署到大量的机器集群上，同时要求不中断和破坏当前服务。这些产品在发布前需要通过相关测试，发布后则要求实时监控，支持故障转移、服务降级、快速定位和故障修复。但是在传统的开发运维流程下：

- 运维人员可能对应用程序内部缺乏了解，难以正确快速地配置环境和发布应用。
- 开发测试环境和真实生产环境不同，运维人员需要修改部署脚本和配置文件来适应生产环境，这有可能引入新的问题，延缓产品的部署。
- 开发人员可能对生产环境缺乏了解，从而难以优化代码及配置，造成应用在生产环境下达不到预期运行的效果。
- 开发人员通常关注的是与业务需求直接相关的功能，而没有考虑监控、故障定位等运维需要的功能。一旦应用在生产环境下发生故障，运维人员无法及时采取措施来恢复运行。
- 开发人员对配置和环境做了修改后，没有及时与运维人员沟通，经常造成新代码无法在产品环境下运行，产品无法及时发布。

基于以上原因，近年来在软件开发领域一个新概念DevOps（Development和Operations的组合）开始流行。DevOps是一种重视软件开发过程中各个团队之间沟通合作的文化、运动或惯例，通过自动化的流程，使得开发、构建、测试、发布软件能够更加快捷、频繁

和可靠，如图1-6[①]所示。

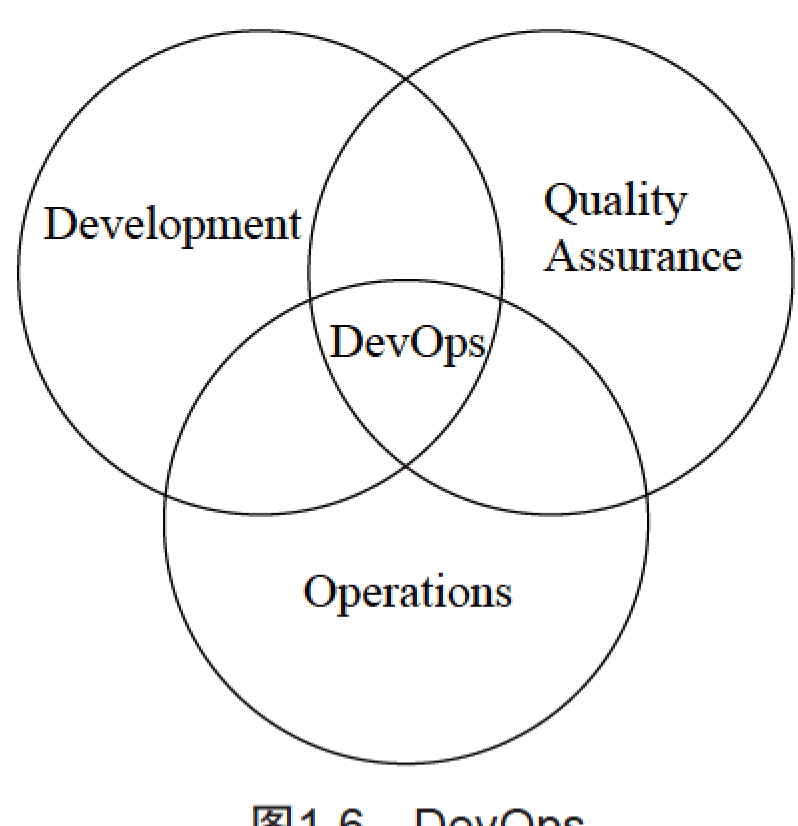


图1-6　DevOps

DevOps不是一种技术或工具，而是一种文化转变，它鼓励团队合作，以便更快地构建可靠性更高、质量更好的软件：

- 运维人员要懂得产品的架构与功能，而不仅仅是管理员手册。
- 开发测试人员要懂实际的运维，包括从实际部署、上线流程，到故障的定位与解决。
- 运维所需的功能甚至基础设施要成为产品非功能性需求的一部分。
- 产品交付与运维需要集成到整个软件生命周期中。

从瀑布模型到敏捷开发，敏捷开发到持续集成，持续集成到DevOps，不管流程如何制定，目的都是相同的：在不牺牲质量的情况下更快地交付产品。

1.5　本书目标

对Web前端应用进行测试之所以困难，一方面是因为代码运行的环境几乎无法控制，各种类型的操作系统，各种版本的操作系统，各种类型的浏览器，各种版本的浏览器，各种语言、插件、扩展，各种前端框架都交织在一起；另一方面其快速迭代的特性使得测试周期短，测试质量难以保证。

同时，Web前端应用开发模式的转型也给开发人员和测试人员带来了新的挑战。开发人员需要了解一定的测试方法和思维才能设计出良好的测试用例。而测试人员需要一定的

① Wikipedia. DevOps[OL]. [2016]. https://en.wikipedia.org/wiki/DevOps.

技术能力才能充分利用自动化测试工具提高测试效率。由于测试和开发进行了融合，所以无论是开发人员还是测试人员，都需要不断提高自身的能力和价值。

本书目标是通过介绍：

- 单元测试
- 自动化测试
- 持续集成

使读者了解如何利用各种工具框架编写测试用例，对Web前端应用进行高效测试，并最终提高软件产品的质量。

第2章 搭建测试基础环境

编写和运行本书的示例代码需要相应的工具和运行环境。本章将介绍测试基础环境的搭建和常用工具。

本章将介绍：

- JavaScript的运行环境Node.js
- 软件包管理系统Node Package Manager（npm）
- 代码编辑器（Visual Studio Code）

2.1 JavaScript的运行环境Node.js

2.1.1 什么是Node.js

Node.js这个名字很容易让人以为是一个JavaScript的应用或类库。实际上，Node.js是一个JavaScript的运行环境，主要采用C++语言编写而成[①]。

要了解Node.js，首先要了解什么是JavaScript引擎。JavaScript是一门高级语言，计算机并不能直接执行，所以需要使用所谓的引擎来将其转换成计算机能理解的机器语言。最初，JavaScript主要运行在浏览器里，浏览器中的JavaScript引擎负责解析和执行网页中的JavaScript代码，并提供代码执行的运行环境，例如内存管理（内存分配，垃圾回收）、即时编译（Just-in-time Compilation）、类型系统（Type System）等服务。

C#和JavaScript的运行环境对比，如表2-1所示。

---

① Ryan Dahl. node.js[OL]. 2009. http://s3.amazonaws.com/four.livejournal/20091117/jsconf.pdf.

表2-1 C#和JavaScript的运行环境比较

C#的运行环境	JavaScript的运行环境
.Net的Common Language Runtime (CLR)	浏览器的JavaScript引擎（Chrome/V8、IE/Chakra、Firefox/*Monkey）

如果能把JavaScript引擎从浏览器中独立出来，那么JavaScript代码就可以被移植到浏览器以外的环境中执行，和其他高级编程语言一样做网页交互以外的事情，从而大大拓宽了JavaScript的应用范围。

2008年，Google公司为Chrome浏览器开发了开源JavaScript引擎V8。Node.js的诞生可以说很大程度上归功于V8引擎的出现。当时其他JavaScript引擎对JavaScript代码进行解释执行，性能较差；而V8使用即时编译，在代码执行前将JavaScript编译成二进制机器码再执行，极大地改善了JavaScript程序的性能，使得JavaScript程序在V8引擎下的运行速度可以媲美二进制程序。

除了能够大幅提升JavaScript性能，V8引擎也可以作为独立的模块，由开发者在自己的C++程序中"嵌入"V8引擎，从而高效地编译JavaScript。

2009年，Ryan Dahl创建了基于V8引擎的Node.js项目（后来得到Joyent公司的资助）。Node.js是一个开源的、跨平台的JavaScript运行环境，最初发布在Linux平台上，直到2011年7月，在微软的支持下才发布了Windows版本。它对V8引擎进行了封装，并提供很多系统级的接口调用，如文件操作、网络编程等，可以用JavaScript编写响应速度快、易于扩展的网络应用。

Ryan Dahl创造Node.js的目的是为了实现高性能的Web服务器。在传统服务器软件开发中，并发的请求处理一直是个大问题。传统服务器模型通常为每一个请求生成一个新线程或新进程，一方面服务器创建新线程/新进程会造成延时，另外一方面，新线程/新进程会消耗额外的内存，浪费资源。在这种传统模型中，如果应用程序的某个任务很耗时，涉及到大量I/O操作（比如访问文件），对应的线程就处于一种不占用CPU，而只是等待响应的状态，直到数据传输完成。但由于等待期间该线程/进程依然占用着资源，当大量并发请求到达时，就会产生阻塞，造成服务器瓶颈。

Node.js以事件驱动为核心, 使用非阻塞I/O模型，它为每个连接发出（emit）一个事件（event），放进事件队列当中，而不是为每个连接生成一个新的线程/进程。理论上，只要有用户请求连接，Node.js都可以进行响应。同时，该I/O模型提供的绝大多数应用编程接口都是基于事件的、异步的风格。开发人员根据自己的业务逻辑需要在相应的事件上注册回调函数，这些回调函数在相应事件触发后被调用。例如，当应用程序发生一个I/O操作（比如访问文件）时，Node.js的主线程可以继续执行，而无需等待这个操作完成；等到

这个I/O操作完成，相应事件被触发的，回调函数被再执行。这使得Node.js在相对较低的系统资源消耗下具有高性能与出众的高并发能力。

JavaScript是Node.js的天然载体语言，因为它允许使用匿名函数和闭包，非常适合事件驱动和异步编程。并且JavaScript作为一门编程语言自身不带I/O功能（一般的编程语言都带一个I/O模块，很不幸的是这个模块通常是同步的，而JavaScript最初是在浏览器内运行，浏览器负责I/O，所以JavaScript没有这个历史包袱）。

Node.js自诞生以来发展迅速，社区活跃，吸引全世界开发者为其贡献了大量的工具、模块和框架。很多企业也逐渐开始采用Node.js开发项目。

本书虽然不是介绍如何利用Node.js开发高性能Web应用的，但是示例所需的很多工具和框架依赖于Node.js的JavaScript运行环境，所以需要安装Node.js。

2.1.2　Node.js的版本发展

初次接触Node.js的读者，可能会对Node.js的版本产生困惑。你可能看到过v0.8.x、v0.10.x、v0.11.x、v0.12.x等版本，然后版本号突然变成4.x.x（本书编写时最新版本号是6.8.1），这和Node.js的发展历史有很大的关系。

Ryan Dahl创建Node.js时，采用的是和Linux内核相同的奇偶版本模式，即版本由3个整数组成，格式为“A.B.C”，A代表主版本号，B代表次版本号，C代表较小的末版本号。只有在内核发生很大变化时，A才变化。C代表一些缺陷修复、安全更新、新特性和驱动的次数。而通过数字B来判断产品是否稳定，偶数的B代表稳定版，奇数的B代表不稳定的开发版。

2010年，Joyent公司雇用了Ryan Dahl并让其专职负责Node.js的发展，在此同时，还得到了Node的品牌使用权[①]。Joyent的Node.js继续使用奇偶版本模式，比如v0.8.x、v0.10.x、v0.11.x、v0.12.x。

2012年，Ryan Dahl离开了Node.js的项目负责岗位并淡出了公众视野。Ryan Dahl离开后，Node.js 开源社区的贡献者和 Joyent 发布的更新数量都在不断缩减。

2014年12月，由于对Joyent公司垄断Node.js项目以及该项目进展缓慢的不满，一部分核心开发者离开了Node.js，创造了io.js项目。这是一个更开放、更新更频繁的Node.js版本。io.js的版本策略是语义化版本（Semantic Versioning）[②]，使用3个整数表示向后兼容的

---

① Ryan Dahl. Joyent & Node[OL]. 2010. https://groups.google.com/d/msg/nodejs/lWo0MbHZ6Tc/6z2C45u6mpAJ.

② Tom Preston-Werner. Semantic Versioning 2.0.0[OL]. [2016-10-19]. http://semver.org.

程度，格式为MAJOR.MINOR.PATCH：

- MAJOR（主要）。此位版本号表示此版本号和前一个版本号在API上不兼容，如3.0.0的API可能不兼容2.0.0的请求。
- MINOR（次要）。此位版本号表示新添加了一个功能，且保持旧功能完全向后兼容。
- PATCH（补丁）。此位版本号表示修复了以前功能的Bug，且完全向后兼容。

2015年2月，Joyent公司宣布放弃对Node.js项目的控制，将其转交给新成立的开放性质的Node.js基金会。

2015年9月，io.js项目宣布回归Node.js。Joyent的Node.js v0.12.7 和io.js 的v3.3.1合并成新的Node v4.0，这是为了防止与后续Joyent的Node.js 0.x.x 维护计划和任何现有的io.js版本发生冲突。这次合并之后，整个项目采用语义化版本编号策略。所以4.0.0 是重生后的Node.js真正的“1.0”版本。

注意，如果读者使用的是Visual Studio 2015，安装时可能选装Joyent Node.js，如图2-1所示，但是它的版本是v0.12.2。所以本书建议采用从官网下载安装包的方式安装Node.js。

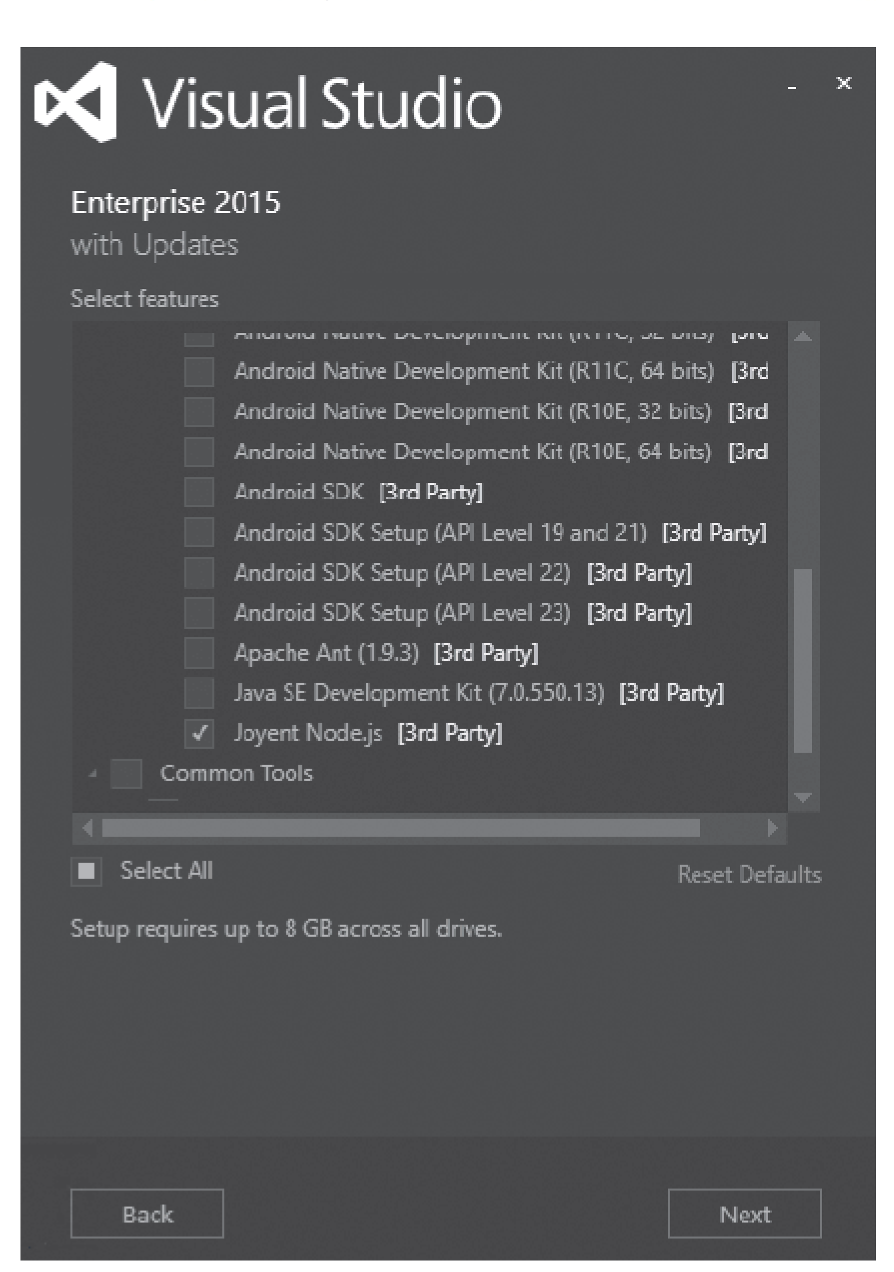


图2-1　Visual Studio 2015安装界面

2.1.3　安装Node.js

Node.js可以在主流操作系统上运行，包括 Windows、Linux和Mac OS。Node.js安装包

及源码下载地址为https://nodejs.org/en/download/，如图2-2所示。

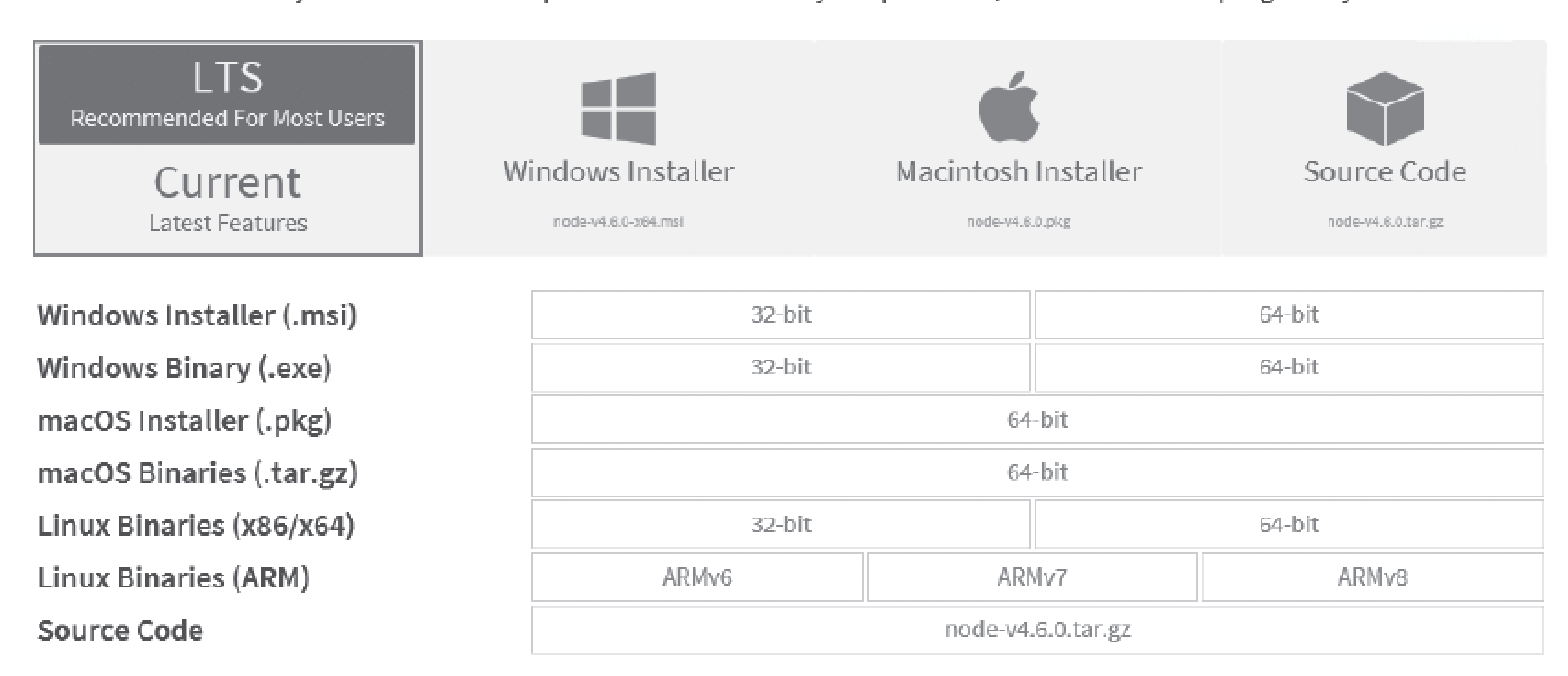


图2-2　Node.js下载页面

用户可以根据不同操作系统选择所需要的Node.js安装包。本书示例基于Windows 10操作系统，采用Node.js v4.6.0 LTS（长期支持版本）的64位Windows安装包（.msi）来安装Node.js，步骤如下：

（1）双击下载后的安装包node-v4.6.0-x64.msi，出现欢迎界面，单击Next按钮。

（2）勾选接受最终用户许可协议选项，单击Next按钮，如图2-3所示。

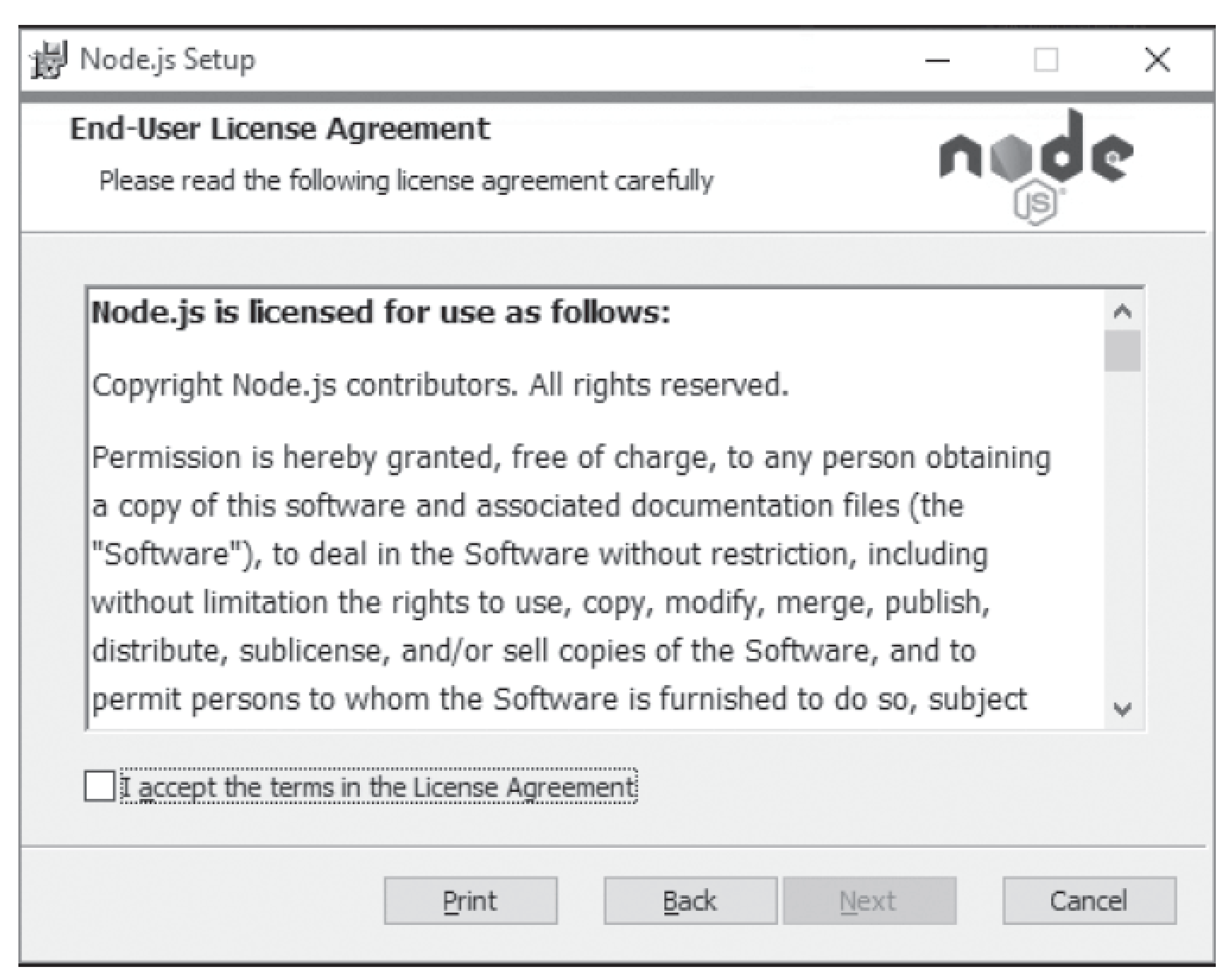


图2-3　Node.js用户许可协议

（3）使用默认安装目录和默认安装方式安装Node.js。安装完毕后单击Finish按钮退出安装向导，如图2-4所示。

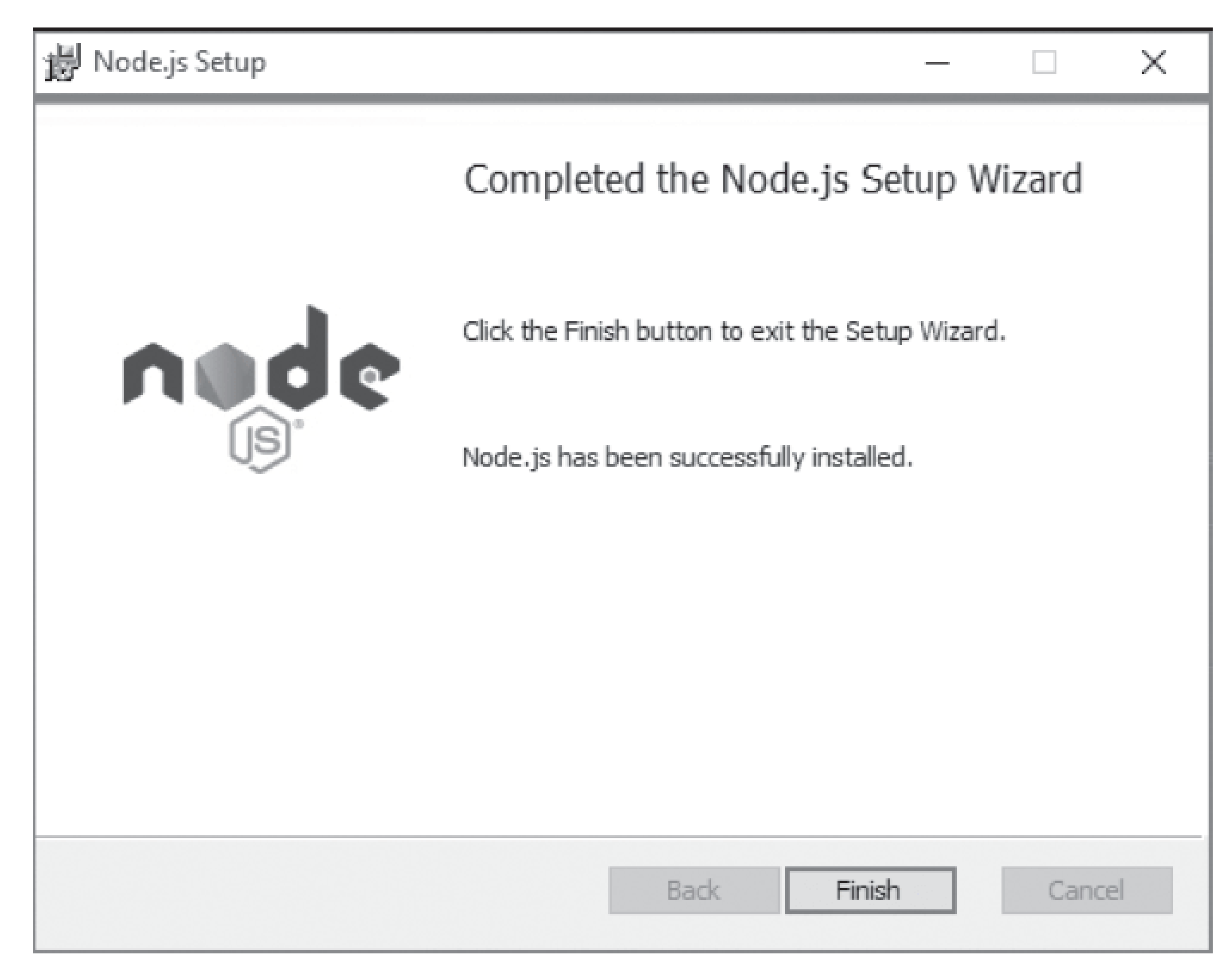


图2-4　Node.js安装完成

打开命令控制台，运行以下命令检查当前Node.js的版本：

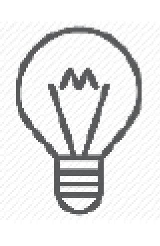
```
C:\>node --version
v4.6.0
```

为了验证Node.js能否正常工作，可以直接输入node，按回车键。

```
C:\>node
>
```

如果Node.js成功安装，此时就进入了Node.js的命令行模式，可以直接输入JavaScript命令，按回车键执行。例如，执行命令console.log('Hello World!')：

```
C:\>node
>console.log('Hello World')
Hello World!
Undefined
```



采用以下两种方法可以退出Node.js的命令行模式：

- 按两次快捷键Ctrl+C。
- 输入.exit，按回车键。

2.2 软件包管理系统Node Package Manager（npm）

依赖关系是软件包管理的一个重要方面。每个软件包都有可能依赖其他的软件包，例如软件包A需要软件包B，而软件包B需要软件包C。通常这些软件包并不具备自动安装所依赖的软件包的功能，当用户安装软件包A时，他需要预先手工安装软件包B和C。如果依赖关系复杂的话，则用户将不得不花大量精力去处理这些软件包之间的依赖关系。

软件包管理系统一般能够从软件源处自动下载所依赖的软件包并安装，解决软件包之间的依赖关系，所以软件包管理系统在各种系统软件和应用软件的安装管理中均有广泛应用。例如，微软.Net开发平台上的软件包管理系统是NuGet。

Node Package Manager（npm）顾名思义是Node.js的软件包管理系统，完全以JavaScript编写，支持跨平台，由Isaac Z. Schlueter在2010年创建，主要功能包括：

- 一个在线仓库（https://registry.npmjs.org），允许开发人员将自己编写的JavaScript软件包注册并上传到这个在线仓库供下载使用。
- 命令行工具用于解决JavaScript软件包的依赖关系，例如从在线仓库中搜索、下载、安装、卸载、更新所需要的JavaScript软件包，并将它们整合到自己的项目中。在本书中npm主要指这个命令行工具。

2.2.1 安装和更新npm

npm不需要单独安装，安装Node.js时会一并安装npm，它是Node.js的默认软件包管理工具。由于npm更新频繁，Node.js附带的npm可能不是最新版本，用户可以在命令控制台执行以下命令将其更新到最新版本。

```
npm install npm@latest -g
```

运行npm命令可查看各种信息。

（1）查看npm的版本，命令如下：

```
npm -v
```

（2）查看npm命令列表，命令如下：

```
npm help
```

（3）查看npm的配置，命令如下：

```
npm config list -l
```

2.2.2 package.json

Node.js项目的根目录下一般会有一个package.json文件。这个文件定义了当前项目的属性，包括项目运行时所依赖的软件包。

package.json常用属性如表2-2所示。

表2-2 package.json常用属性

属性名称	描　　述
name	项目名称（全部小写，没有空格，允许使用下画线和减号）
description	项目描述
version	版本号（语义化版本）
repository	资源仓库地址
main	项目入口文件
scripts	脚本
licenses	授权方式
dependencies	项目运行所依赖的软件包
devDependencies	开发环境所依赖的软件包

以下是一个基本的package.json文件。在package.json文件中，只有name和version字段是必要的，其他字段都是可选的。

```
{
  "name": "myproj",
  "version": "1.0.0",
```

```
  "description": "",
  "main": "index.js",
  "dependencies": {
    "jquery": "^3.1.1",
  },
  "devDependencies": {
    "karma": "^1.3.0"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

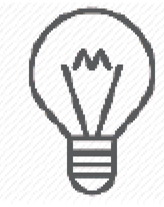
以上package.json示例定义了项目的入口文件为index.js。当其他Node.js应用通过require引用这个软件包时，index.js文件将被调用。

package.json文件可以手工编写，也可以通过执行npm init命令自动生成。该命令采用互动方式，要求用户回答一些问题，然后在当前目录下生成一个基本的package.json文件。所有问题之中，只有项目名称（name）和项目版本（version）是必填的。

本书的示例是Web前端程序，不是Node.js项目，那为什么要介绍package.json文件呢？

2.2.3　安装软件包

本书介绍package.json文件，是因为它可以定义项目运行时所依赖的软件包（dependencies）和开发环境所依赖的软件包（devDependencies）。有了package.json文件，开发人员可以在项目根目录下直接执行npm install命令，该命令会根据package.json文件从在线仓库中下载dependencies和devDependencies中列出的软件包，安装到当前目录的node_modules子目录中，这样就解决了Web前端程序的依赖问题。



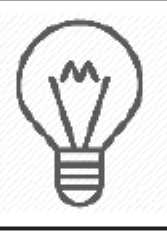
package.json文件里的dependencies和devDependencies字段列出的软件包采用语义化版本，npm install会根据这个信息下载相应版本的软件包，例如：

```
"devDependencies": {
  "karma": "^1.3.0"
},
```

其中，^前缀表示与指定版本兼容，最左边的主要版本不变，但是次要版本和补丁版本可以是更高的版本。例如“^1.3.0” 表示这个项目支持>=1.3.0但是<2.0.0版本的Karma。npm install会下载符合条件的最新版本的软件包。如果想要了解更多有关npm的语义化版本信息，可以参考https://docs.npmjs.com/misc/semver。

如果只安装dependencies字段里列出的软件包，可以执行以下命令：

```
npm install --production
```



不要将node_modules目录提交到源代码管理系统中。基于package.json，只需要执行命令npm install就可以恢复项目的开发和运行环境。

如果项目依赖的软件包没有被定义在package.json文件里，那么这些软件包需要单独安装。安装软件包的形式分两种：本地安装与全局安装。

1. 本地安装

本地安装软件包的命令是：

```
npm install <package name>
```

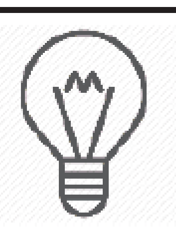
本地安装的软件包会被下载到当前所在目录（通常是当前项目的根目录）的node_modules子目录中，适用于安装一些JavaScript库和框架。这些库和框架会被当前项目所引用。

如果使用--save参数，npm命令会将软件包信息写入package.json文件的dependencies字段中，这通常用于安装项目运行所需要的软件包。

```
npm install <package name> --save
```

如果使用--save-dev参数，npm命令会将软件包信息写入package.json文件的devDependencies字段中。这样的软件包通常仅被用于开发环境。

```
npm install <package name> --save-dev
```



建议在每个项目的根目录中都创建一个package.json文件。

在使用npm安装软件包时，建议使用--save 或--save-dev参数。

例如当前项目需要karma软件包，c:\myproj是当前项目的根目录，打开命令控制台，将当前目录切换到c:\myproj，执行以下命令：

```
C:\>cd c:\myproj
```

然后执行以下命令（因为karma用于测试JavaScript代码，属于开发环境需要的软件包，所以使用--save-dev参数）：

```
C:\myproj>npm install karma --save-dev
```

安装完毕后会输出以下信息：

```
C:\myproj>npm install karma --save-dev
myproj@1.0.0 C:\myproj
`-- karma@1.3.0
  +-- bluebird@3.4.6
  +-- body-parser@1.15.2
  | +-- bytes@2.4.0
  | +-- content-type@1.0.2
  | +-- debug@2.2.0
  | | `-- ms@0.7.1
  | +-- depd@1.1.0
…
```

以上输出信息的简单说明如下：

- karma@1.3.0，当前安装的软件包，版本为1.3.0。

- bluebird@3.4.6，karma依赖的软件包，版本为3.4.6。
- body-parser@1.15.2，karma依赖的软件包，版本为1.15.2。body-parser依赖的软件包在这个树状结构的下一层。

karma软件包被安装在node_modules目录下的karma子目录中。除karma目录外，还有其他依赖软件包的目录，这些都是NPM根据软件包的依赖关系自动下载的，如图2-5所示。

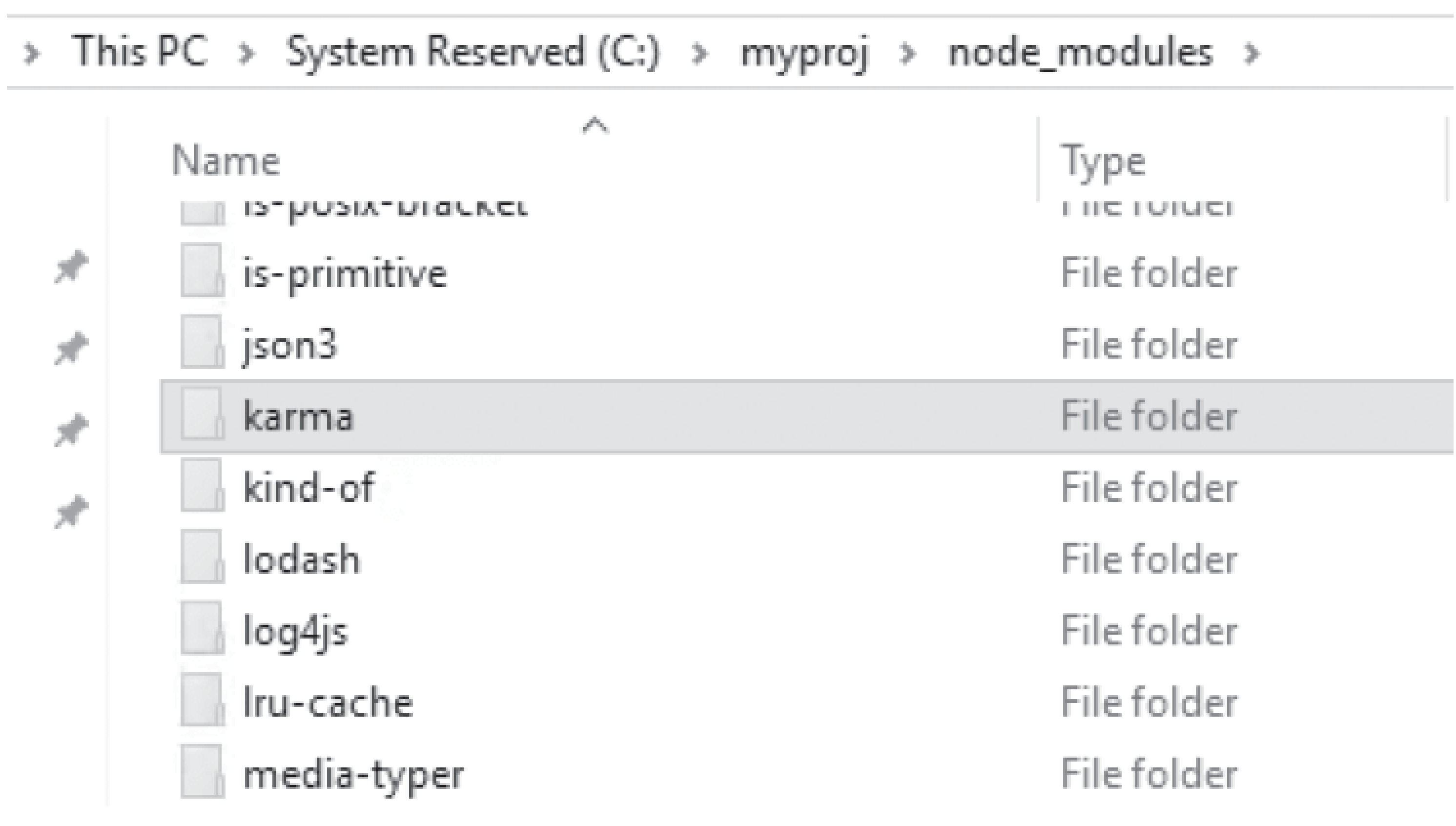


图2-5　node_modules目录

2. 全局安装

全局安装指的是将软件包安装到一个全局安装目录中，这样各个项目都可以使用这些软件包。一般来说，全局安装适用于各种Node.js工具。例如npm本身就是一个全局安装的软件包，所以可以在命令控制台里直接执行npm。

全局安装软件包的命令是：

```
npm install -g <package name>
```

例如，本书示例需要使用gulp工具，就可以采用全局安装的方式，使各个项目都可以使用gulp工具。具体命令如下：

```
npm install -g gulp
```

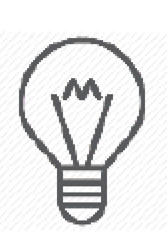
运行以上命令后，gulp工具被安装到目录{prefix}\node_modules\中。在作者的计算机上，{prefix}是C:\Users\demouser\AppData\Roaming\npm目录。以下命令会显示当前{prefix}的值：

```
npm config get prefix
```

全局安装目录也可以通过以下命令进行修改：

```
npm config set prefix=c:\global_packages
```

修改全局安装目录后，软件包会被安装到c:\global_packages\node_modules。



修改完全局安装目录后，请将新的目录添加到Windows的环境变量PATH中，并将旧目录从PATH环境变量中删除。以后就可以在命令控制台中直接执行全局安装的工具了。

2.2.4　列出已安装的软件包

npm list命令以树型结构列出当前项目安装的所有软件包，以及它们依赖的软件包。具体命令如下：

```
npm list
```

如果加上一个参数-g或--global就可以列出全局安装的软件包和它们依赖的软件包。具体命令如下：

```
npm list -g
```

如果不想输出所依赖软件包的信息，可以添加--depth参数。具体命令如下：

```
npm list -g --depth=0
```

运行以上命令将得到如下结果：

```
C:\>npm list -g --depth=0
c:\global_packages
+-- gulp@3.9.1
`-- npm@3.10.8
```

2.3 代码编辑器（Visual Studio Code）

一款好用的编辑器能够显著提升开发人员的工作效率。本书示例使用的是微软开发的跨平台（支持Windows、Linux和Mac OS操作系统）开源代码编辑器Visual Studio Code。

和全功能的集成开发环境Visual Studio不同，Visual Studio Code的定位是一个轻量但又功能强大的代码编辑器，可帮助开发人员进行快速编码、编译和调试。

Visual Studio Code来源于微软的一款使用HTML、CSS和JavaScript开发的在线编辑器Monaco[①]（用于Visual Studio Online、OneDrive等），在这个基础上利用基于io.js和Chromium的开源框架Electron[②]进行包装（Electron可以让用户使用JavaScript调用操作系统的原生API来创造跨平台桌面应用），成为一款跨平台的桌面代码编辑器。

作为代码编辑器，Visual Studio Code支持多种编程语言，其中原生支持JavaScript、TypeScript、CSS和HTML。开发人员可以通过VS Code Marketplace[③]下载扩展插件获得其他编程语言的支持。

Visual Studio Code支持调试。原生调试功能限于Node.js环境，可以调试JavaScript、TypeScript和其他能够被转译为JavaScript的编程语言。其他运行环境和编程语言（例如PHP、Ruby、Go、C#、Python）的调试支持也可以从VS Code Marketplace下载扩展插件获得。

Visual Studio Code内置了Git版本控制功能，支持用户自定义配置，例如改变主题颜色、键盘快捷方式、编辑器属性等。

2.3.1 安装Visual Studio Code

安装Visual Studio Code的方法是，先从官网https://code.visualstudio.com/下载安装文件包，双击VSCodeSetup-stable.exe启动安装程序，如图2-6所示。

然后单击Next，按安装向导指示使用默认设置安装即可。安装结束后可以直接启动Visual Studio Code，如图2-7所示。

---

① Microsoft. A browser based code editor[OL]. [2016]. https://github.com/Microsoft/monaco-editor.

② Electron. Build cross platform desktop apps with JavaScript, HTML, and CSS[OL]. [2016]. http://electron.atom.io.

③ Microsoft. Extensions for the Visual Studio family of products[OL]. [2016]. https://marketplace.visualstudio.com/VSCode.

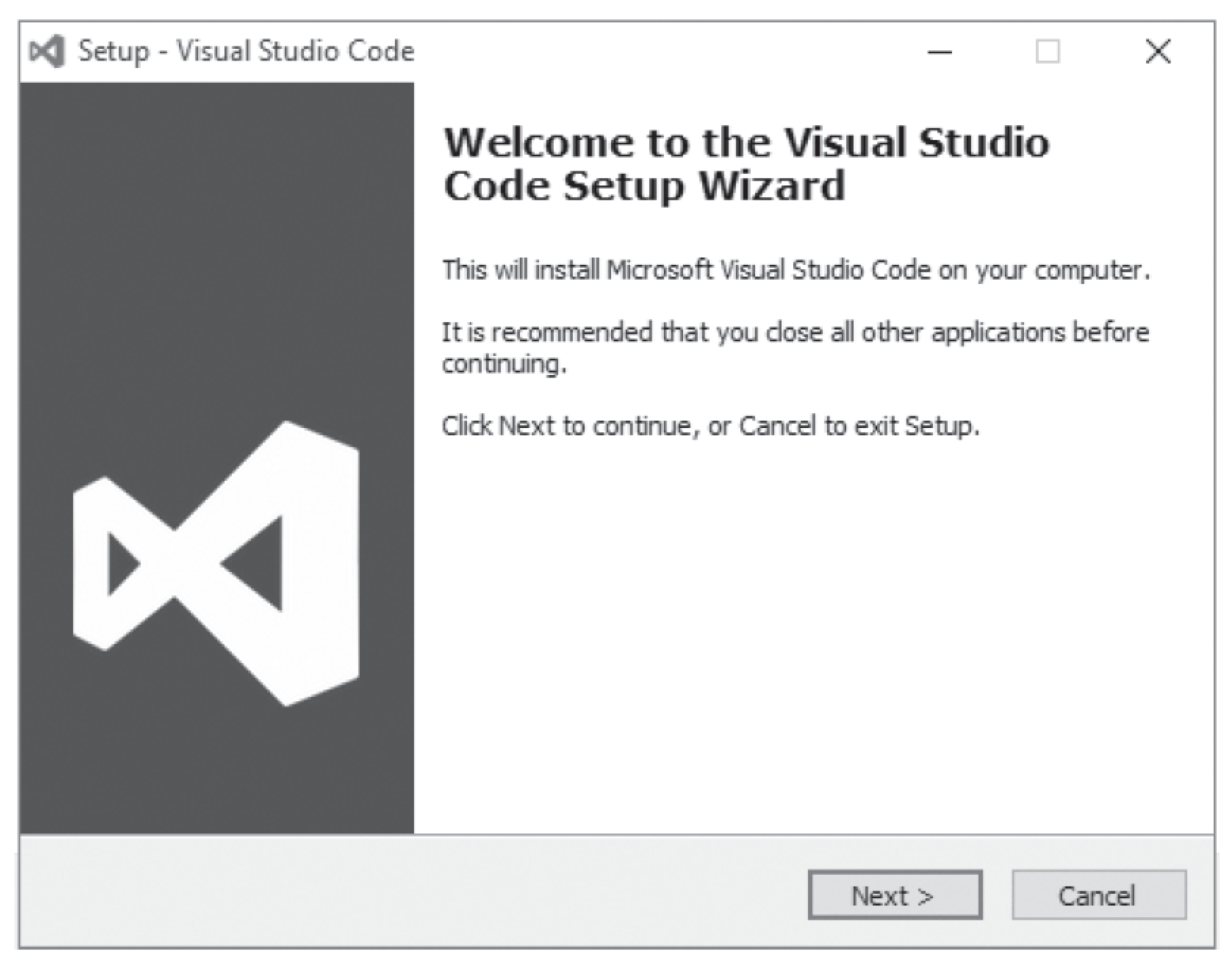


图2-6　Visual Studio Code安装欢迎界面

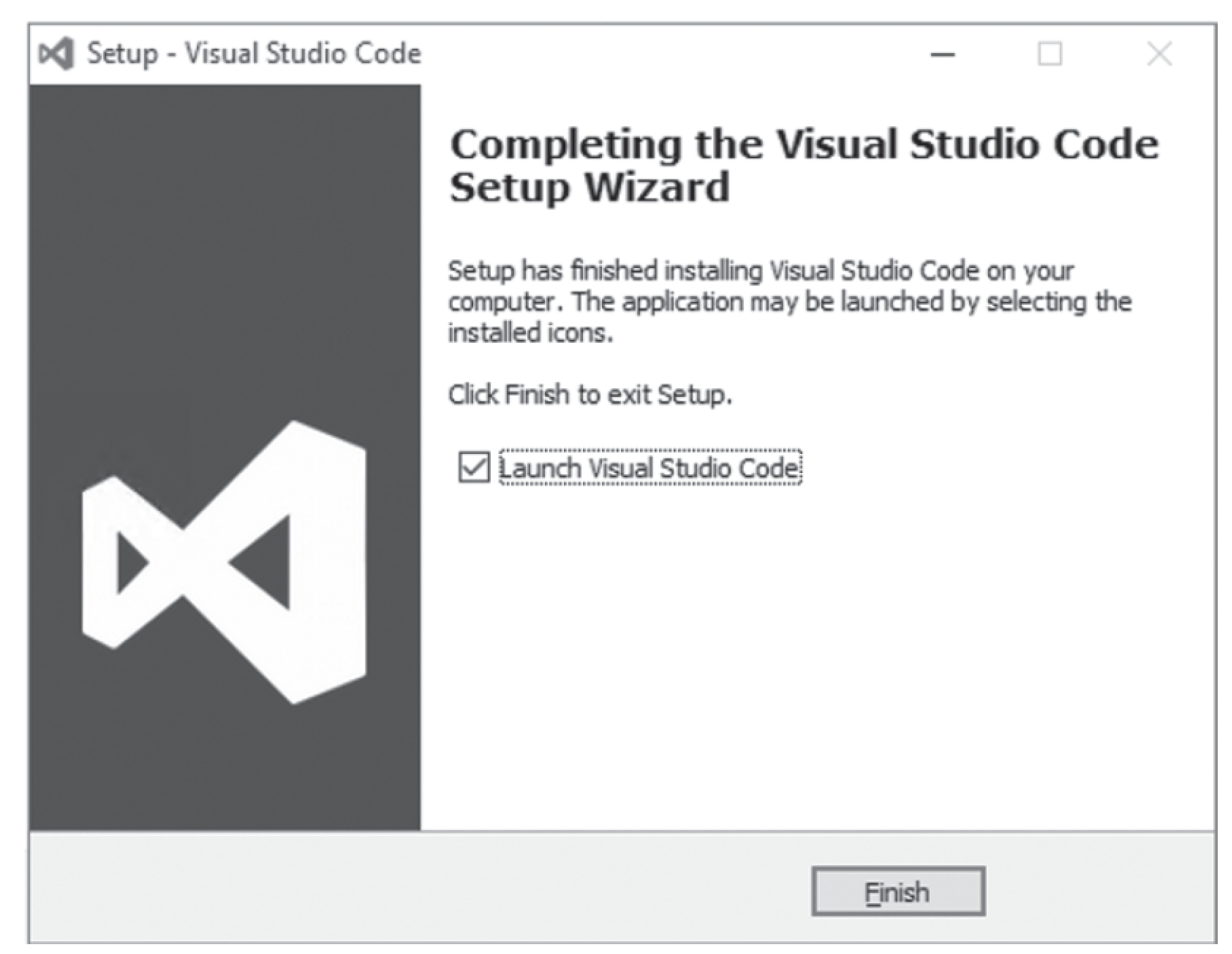


图2-7　Visual Studio Code安装完成

2.3.2　初识Visual Studio Code

和Visual Studio不同，Visual Studio Code管理项目没有专门的项目文件，它是基于文

件和目录的。用户可以用Visual Studio Code打开一个文件或文件夹。

Visual Studio Code的工作界面有着简单直观的布局，提供了最大化的编辑空间，同时为用户留下足够的空间来浏览和访问文件夹，如图2-8所示。

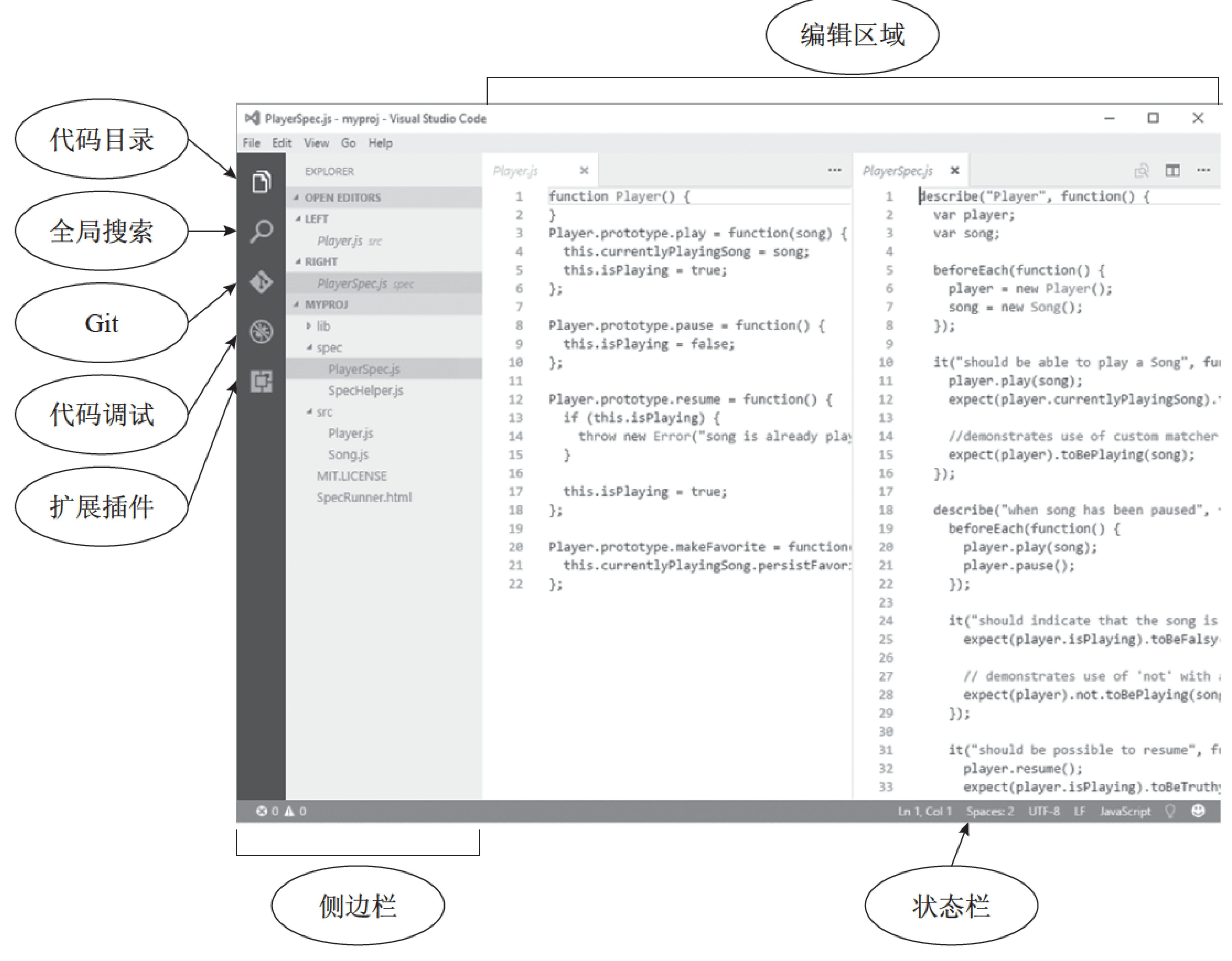


图2-8　Visual Studio Code界面布局

位于工作界面最左边的是视图栏，可用来在代码目录、全局搜索、Git、代码调试和扩展插件这5个视图间切换。

侧边栏根据视图栏的选择显示不同的视图。例如，当用户选择代码目录视图时，侧边栏显示资源管理器，用来浏览、打开、管理所有文件和文件夹。

状态栏用于显示打开的项目和编辑的文件的相关信息。

用户可以在编辑区域内编辑文件。Visual Studio Code同时支持3个可视编辑器，可以编辑或查看并排在一起的3个文件。在编辑区顶部区域，每个被打开的文件都有对应的选项卡（Tabs）。

Visual Studio Code同样支持大量键盘快捷键操作。使用快捷键Shift+Ctrl+P调出命令面

板，在这里可以访问Visual Studio Code所有的功能，包含最常见的快捷键操作，如图2-9所示。

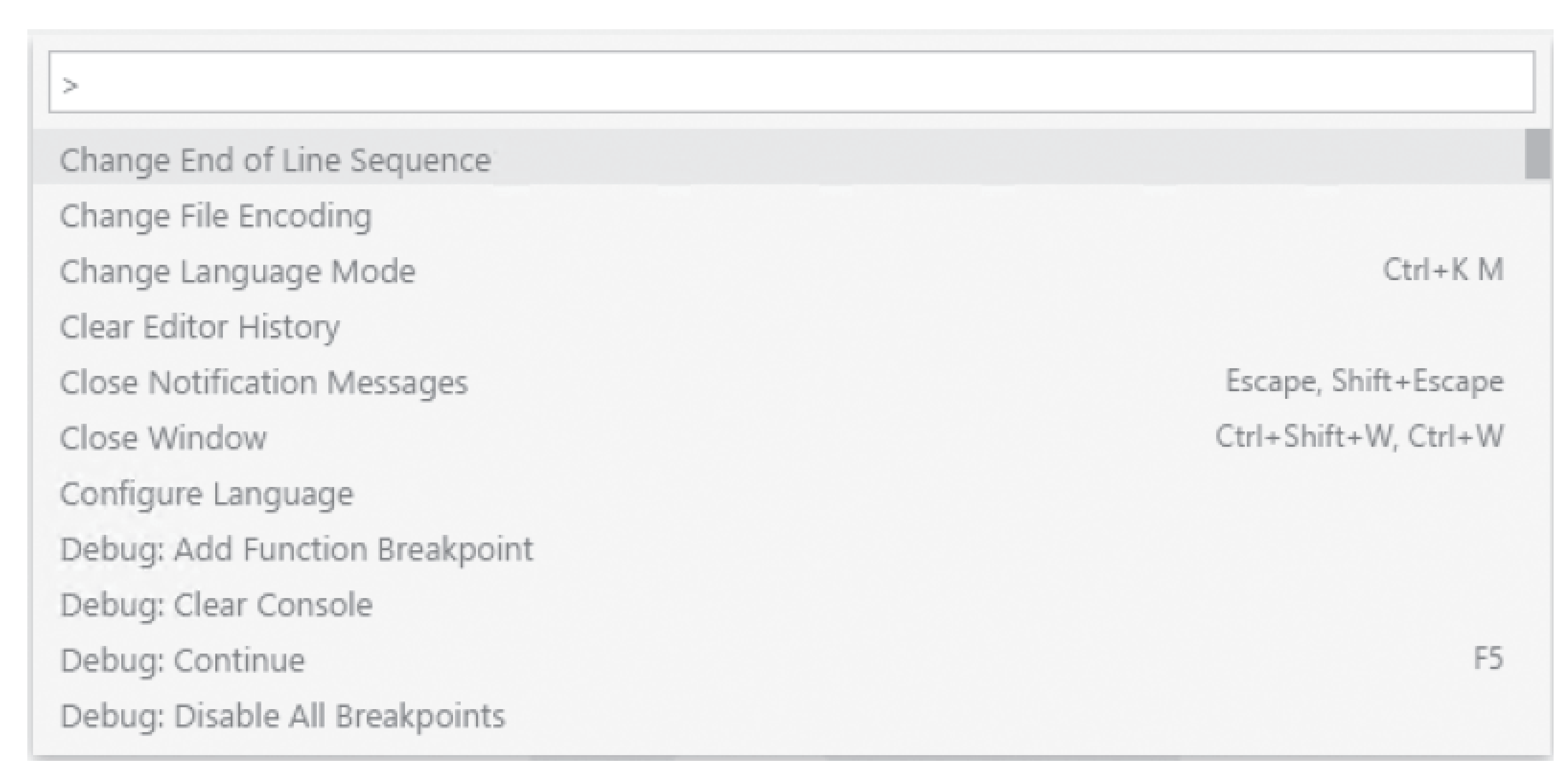


图2-9　Visual Studio Code命令面板

单元测试篇

第3章 单元测试概论

在软件设计领域中有很多种测试，单元测试是其中一种，也是最基本的一种测试。单元测试能够从缺陷的源头找出软件中潜在的问题，它是保证软件质量的重要手段。

本章将介绍：

- 单元测试的特性
- 单元测试的重要性
- 测试金字塔
- 测试先行（Test-First）
- Web前端测试框架

3.1 单元测试的特性

单元测试代码通常由软件开发人员编写。单元测试是针对软件设计的最小单位进行检查和验证的工作，它有以下一些特性：

- 用代码测试代码。单元测试通常是一段测试代码，这段代码调用被测试的程序单元，然后对这个程序单元的单个最终结果的某些假设进行检验。这个过程无须人工干预，如图3-1所示。
- 单元测试本身是代码，可重复自动运行。
- 单元测试针对程序单元，只需要考虑有限的几个情况，编写测试用例通常比较简单。
- 单元测试应该易于安装及运行，它不需要进行烦琐的配置（程序单元已被隔离，它所依赖的部分已经被测试替身代替）。如果单元测试需要访问数据库、网络等，这个测试就不是真正的单元测试。

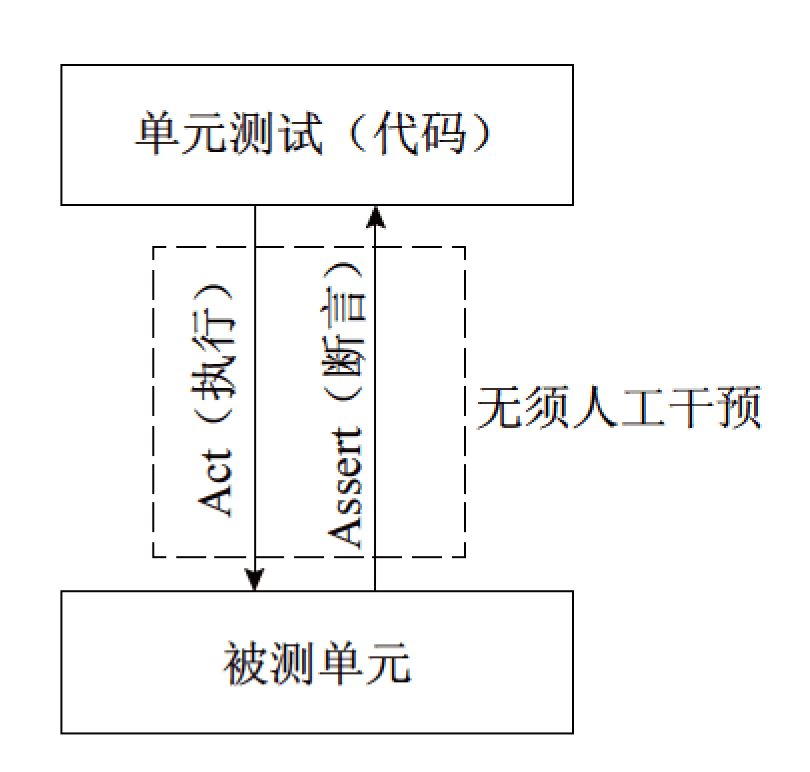


图3-1　单元测试——代码测试代码

- 单元测试的时间应该非常短。通常来说，开发人员每修改一次程序就会进行至少一次单元测试，高效的单元测试可以向开发人员快速反馈信息。
- 在编写程序的过程中通常会进行多次单元测试，所以单元测试在软件开发过程的早期就能发现问题。
- 单元测试一般粒度较小，因此发现了问题，可以快速定位并修复错误。
- 良好设计的单元测试可以覆盖程序单元分支和循环条件的所有路径。

3.2　单元测试的重要性

激烈的商业竞争、飞速发展的新兴技术使得企业需要比以往更快速地交付高质量的软件。因为项目工期紧，任务重，很多开发人员会把主要时间用在编写业务逻辑代码上，有时间的时候才会写一些简单的单元测试代码，甚至不写单元测试。理由如下：

- 没时间写单元测试。
- 单元测试并不能防止漏洞的出现。
- 这段代码很简单，为什么还要写单元测试？

其实，不写单元测试的根本原因还是很多开发人员不了解单元测试所能带来的好处，所以才会认为写单元测试是浪费时间，不去写单元测试。

这里引用J.Timothy King“关于单元测试的十二个好处”[①]。

① J.Timothy King. Twelve Benefits of Writing Unit Tests First[OL]. 2006. http://sd.jtimothyking.com/2006/07/11/twelve-benefits-of-writing-unit-tests-first.

1）单元测试可以证明你的代码真能解决问题

这意味着缺陷减少。单元测试当然不能取代系统测试和验收测试，但能补足它们的短处。

2）可以获得一组底层回归测试

开发人员可以随时检查程序不工作的部分和发现缺陷所在位置。很多团队会每天运行整组单元测试，这样可以在交付程序给质管部门之前及早发现缺陷。

3）可以在不破坏现有功能的基础上持续改进设计

一旦有了单元测试，开发人员就可以很方便地重构代码。只要在重构以后重新运行单元测试就可以知道是不是破坏了现有功能。

4）边写单元测试边写代码是更有趣的工作方式

通过编写单元测试，开发人员能更好地理解代码需要实现的功能。在实际系统没有完成的情况下，可以通过单元测试运行编写的代码。代码的成功运行会给开发人员带来成就感，激励他们去完成后面更多的工作。

5）可以真实地反映开发进度

因为代码可以在实际系统没有完成的情况下运行（在单元测试的帮助下），所以开发人员可以随时展示他们的进度。而且，因为有单元测试，所谓的“编码完成”不仅仅是写完代码，签入代码库，还能够无缺陷地运行。

6）单元测试是一种使用范例

我们都碰到过那种不知道该怎么用的函数或类，这种情况下一般会先去找范例代码，但是通常内部使用的代码不会有范例。幸运的是单元测试可以作为一种文档，当不知道某个函数怎么使用时，看一下单元测试代码怎么写即可。

7）促使写代码前先做规划

先写测试（后写代码）会促使开发人员在动手开发前把必须完成的事和整体设计考虑一遍。这不但会让他们更专注，还能让设计更漂亮。（本章后面会介绍测试驱动开发的相关内容）

8）降低缺陷修复成本

缺陷发现得越早越容易修复。发现得晚的缺陷通常是好几处代码变动引发的结果，而且往往不知道究竟是由哪个变动造成的，这使得修复缺陷变得相当困难。单元测试能帮助开发人员及早发现缺陷。

9）单元测试甚至比代码审查的效果还要好

有人说事前代码审查比事后测试更好，因为发现和修复缺陷的成本更低。如果代码发

布后才去修复缺陷，成本就高得多。而单元测试能够比代码审查更早地发现缺陷。

10）无形中为开发人员消除了工作上的障碍

开发人员在编码时可能会碰到无法继续工作的障碍。单元测试能够系统化代码结构，帮助开发人员将注意力集中到创造新功能上。他们可能卡在不知道如何测试下一段代码，或者不知道如何让测试通过，但他们永远知道下一步该做什么。有时候你很想在累倒之前休息一下，但是因为工作进行得很顺，使得你根本不想停下来。

11）单元测试促成更好的设计

为了测试一段代码，开发人员需要清晰地定义代码的功能。如果代码测起来很简单，这表示这段代码功能清晰，具有很高的聚合度。如果代码能通过单元测试，说明它很容易被整合到实际系统中，和周边代码具有松耦合的依赖关系。高内聚和松耦合往往是优秀设计的标志。同时容易通过单元测试的代码也容易维护。

12）编写单元测试会使开发效率更高

编写单元测试会使开发效率更高。换句话说，忽略单元测试也许能更快完成编码，但是无法保证代码能真正工作，以后开发人员可能不得不花费大量精力去修复代码缺陷。而单元测试能够从源头发现问题，快速修复缺陷。

综上所述，单元测试很重要，和写功能代码一样重要。

3.3　测试金字塔

传统的软件测试流程一般是先在软件开发过程中进行少量的单元测试，然后在整个软件开发结束阶段，集中进行大量的测试，包括集成测试和端到端测试。随着软件项目越来越复杂，大量的错误往往只有到了项目后期端到端测试时才能够被发现，项目进度和项目风险难以控制。错误发现得越晚，错误修复成本越高。错误的延迟解决必然导致整个项目成本的急剧增加。

为此需要改变测试方法，在各种测试之间找到正确的平衡，把时间用在正确的测试活动中，尽可能避免上述问题，这就是测试金字塔。

测试金字塔的概念来自Mike Cohn，在*Succeeding With Agile*一书中有详细描述，其核心观念是底层单元测试应多于依赖UI的高层端到端测试，如图3-2所示。

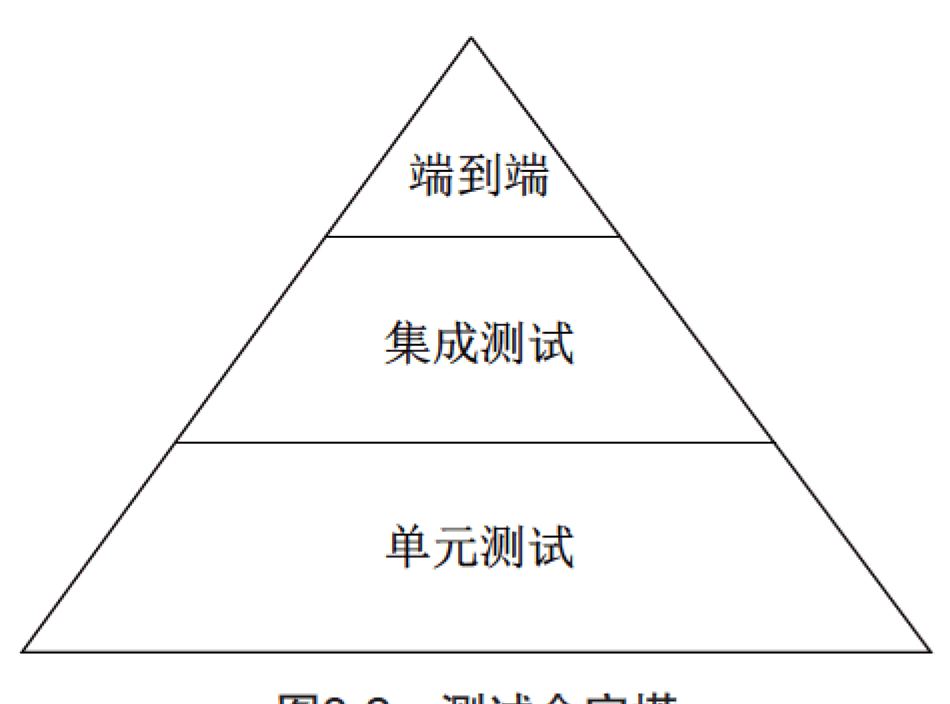


图3-2　测试金字塔

在测试金字塔中，从上往下，成本越来越低，效率越来越高，更贴近技术实现；从下往上，则成本越来越高，效率越来越低，但更接近真实业务需求。测试金字塔理论强调建立一个合理的测试组合，尽可能使用大量低成本的单元测试，辅之以少量高成本但更接近业务的UI端到端测试。

Martin Fowler在他的博客中对测试金字塔进行了如下的解释①：

“特别地，我始终认为高层测试只是测试防护体系的第二防线。如果一个高层测试失败了，不仅仅表明功能代码中存在缺陷，还意味着单元测试的欠缺。因此，无论何时修复失败的端到端测试，都应该同时添加相应的单元测试。”

通常在一个测试组合中，Google测试团队建议采用70/20/10划分②：70%单元测试，20%集成测试和10%端到端测试。实际比例在各个团队中会有不同，但一般来说，会保留金字塔的形状，而尽量避免相反的倒金字塔模式。

3.4　测试先行（Test-First）

尽早发现软件缺陷，并采取合理的应对策略，可以降低整个软件的开发成本。如果在进行软件开发时能够首先编写测试代码（Test-First），也就是说在明确要开发某个功能后，首先思考如何对这个功能进行测试，并完成测试代码的编写，然后编写相关的功能代码满足这些测试用例，那么就会迫使开发人员从易用性、易测试性的角度考虑问题，从而定义出清晰的、明确反映意图的模块接口来。另外，为了使得测试能够通过，开发人员就

① Martin Fowler. TestPyramid[OL]. 2012. http://martinfowler.com/bliki/TestPyramid.html.

② Mike Wacker. Just Say No to More End-to-End Tests[OL]. 2015. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html.

会主动把那些难以测试的耦合（依赖关系）去掉。这样不仅仅是获得了可测试性，而且也产生了更好的设计和系统架构。

3.4.1　测试驱动开发（Test–Driven Development）

Test-Driven Development（TDD）是一种一切软件开发活动都要从首先编写测试代码开始的软件开发过程的应用方法，是敏捷软件开发的推荐做法。

TDD的基本思路就是通过测试来推动整个开发的进程，如图3-3所示，有3个步骤：

（1）在写功能代码前，先用测试用例将功能需求描述出来。因为此时还没有功能实现代码，所以执行测试用例失败。很多软件界面会用红色信息表示测试失败（红灯）。

（2）编写功能代码，使前面失败的测试用例通过。很多软件界面会用绿色信息表示测试通过（绿灯）。

（3）重构功能代码，改善设计。

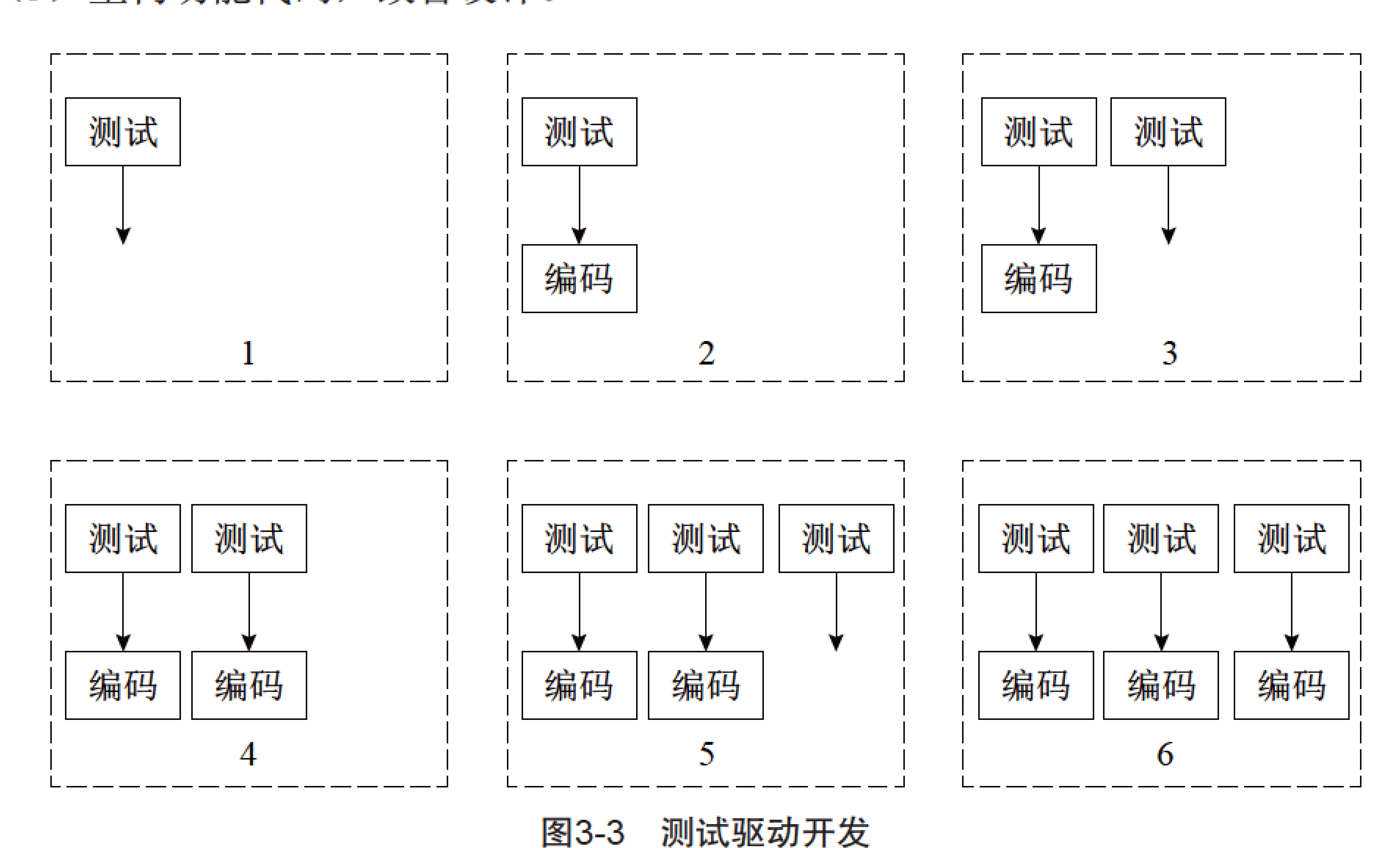


图3-3　测试驱动开发

重复这些步骤：红灯（测试失败）、绿灯（测试通过）、重构，直至全部功能完成。

因为在编写测试用例时，已经对其功能的分解、使用过程、接口都进行了设计，从而将单元测试变成了设计过程的一部分，同时也展示了功能代码是如何工作的。一定程度上，测试用例变成了功能代码的使用文档，实现“代码即文档”的思想。

当然TDD最重要的功能还在于保障代码的正确性，能够迅速发现、定位问题所在。理论上使用TDD永远不会有未被测试的代码，这也给开发人员重构现有代码和对应用进行回归测试（Regression Test）提供了信心和保障。特别是在程序修改比较频繁时，由于测试

用例已经完成，测试期望的结果也是完全可以预料的。

3.4.2 行为驱动开发（Behavior-Driven Development）

Behavior-Driven Development（BDD）是在TDD的基础上发展而来的一种敏捷软件开发的方法。BDD最初是由Dan North在2003年命名，它鼓励软件项目的开发人员、测试人员和非技术人员或商业参与者之间的协作。

BDD作为一种设计方法，本身不是关于测试的，它的重点是客户和技术人员使用几乎近于自然语言的方式（通用语言）描述系统的行为和预期的结果。软件开发中表达不一致是最常见的问题，造成的后果就是开发人员最终做出来的产品不是客户期望的。如果客户（非技术人员）和技术人员使用同一种“语言”来描述同一个系统，则可以最大程度避免表达不一致带来的问题，从而做出符合客户需求的设计。

但是如果光有设计，没有验证的手段，就无法检验实现是不是符合设计，因此BDD还是要和测试结合在一起的。当使用通用语言来描述测试时，可以让任何人更容易地了解被测试的内容。例如，使用“如果银行账户被透支，就不能取款”这样的描述，而不是写一个名为testOverdrawnAccount的测试。

BDD使用用户故事（user story）来描述需求。用户故事通常遵循特定的模板形式：

```
As a [人/角色]
I want [特征/功能]
so that [结果/利益]
```

作为一个[人/角色]，我需要[某些特征和功能]，以便能得到[相应的利益或结果]。

同样的一个故事，可能会有不同的场景。利用以上的模板描述故事之后，可以再通过以下的模板对不同场景进行描述：

```
Scenario :标题（描述场景的单行文字）
Given [上下文]
   And [更多上下文]
When [事件]
Then [结果]
   And [更多结果]
```

以一个经典的ATM取款机为例[1]，故事可以描述为：

```
Story: 银行账户持有人提取现金
As a [银行账户持有人]
I want [从ATM机提取现金]
so that [不需要在银行柜台排队]
```

同样的故事，会有不同的场景发生：

```
Scenario 1: 银行账户有足够资金
Given 银行账户有足够资金
    And 有效的银行卡
    And 提款机有足够的现金
When 账户持有人要求取款
Then 银行账户余额应该被扣除
    And 提款机应该分发现金
    And 应该退还银行卡
```

如果银行账户持有人取款的金额比他的存款还多：

```
Scenario 2: 银行账户透支超过上限
Given 银行账户已透支
    And 有效的银行卡
When 账户持有人要求取款
Then 应该显示拒绝取款的消息
    And 提款机应该拒绝分发现金
    And 应该退还银行卡
```

如果仔细观察，就会发现Given-When-Then其实定义了一个完整的测试，如图3-4所示。

---

① Dan North. Introducing BDD[OL]. 2006. https://dannorth.net/introducing-bdd.

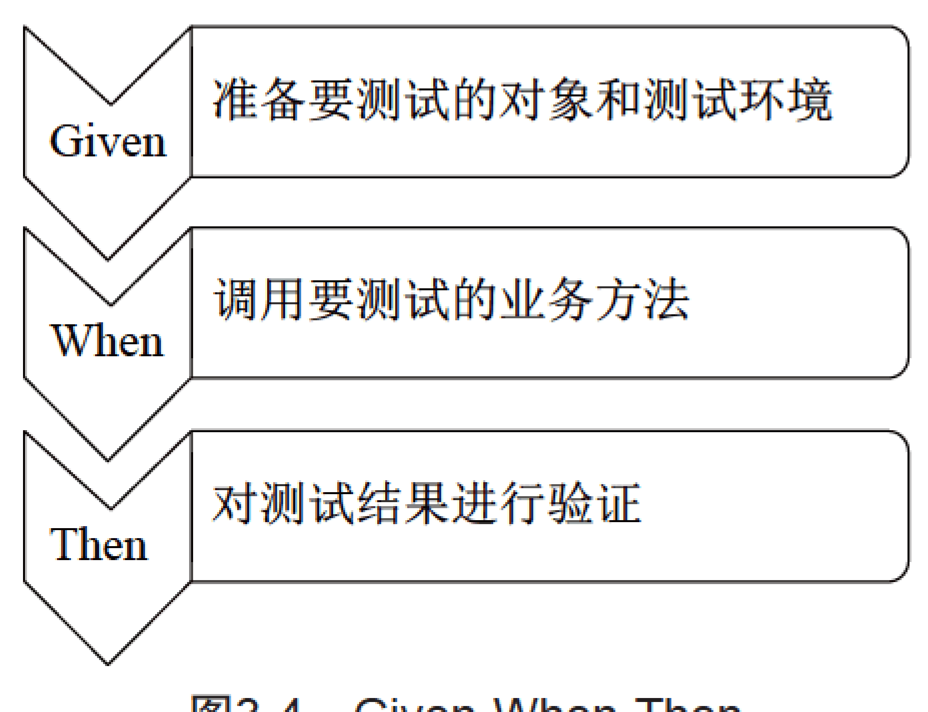


图3-4 Given-When-Then

下一章介绍的Jasmine就是一种BDD风格的JavaScript测试框架。

3.5 Web前端测试框架

工欲善其事，必先利其器。编写单元测试代码之前，需要选择一个测试框架。测试框架是一组测试自动化的规范、基础代码、测试思想的集合，用于组织、管理和执行那些独立的测试用例。同时，测试框架也提供很多方便易用的辅助性工具。使用测试框架可以减少冗余代码，提高代码的生产率、重用性和可维护性。

Web前端JavaScript的测试框架有很多。2012年Google Chrome团队的工程师Addy Osmani曾经在Twitter上做了一次非正式的调研，询问他的粉丝最常用的JavaScript测试框架。其调研结果是[①]：

（1）Jasmine

（2）QUnit

（3）Mocha + Chai

（4）BusterJS

（5）jsTestDriver

（6）CasperJS

目前，Jasmine仍然是最流行的JavaScript测试框架之一。表3-1所示为Jasmine和另外一个主流JavaScript测试框架Mocha做的简单比较。

① Addy Osmani. What JavaScript testing framework do you use the most often?[OL]. 2012. http://bit.ly/JSTestingSurvey.

表3-1　Jasmine和Mocha简单比较

	Jasmine	Mocha
相同点	两者既可以在浏览器里使用，也可以在Node.js环境里使用； 都是BDD风格的测试框架，语法清晰易读； 有相似的API。例如，使用describe函数定义测试套件（test suite），使用it函数定义测试用例（test case）	
不同点	Jasmine是完整的测试框架，自带断言（assertion）库和测试替身（test double）库，无须添加其他依赖； Jasmine没有测试执行器（test runner）。需要第三方测试执行器，例如Karma	Mocha自身不提供断言库和测试替身库； Mocha中可以使用第三方断言库，通常选择Chai； Mocha中可以使用第三方测试替身库，通常选择Sinon.JS； Mocha本身自带测试执行器，也可以使用Karma

本书第4章将深入介绍基于Jasmine的Web前端单元测试。

第4章 深入Jasmine单元测试

Jasmine是什么？Jasmine的作者Davis Frank是这样描述的[①]：

“Jasmine是一个 JavaScript测试框架，目的是将BDD风格引入JavaScript测试之中。至于区别嘛，我们的目标是BDD（相比标准的TDD），因此我们尽力帮助开发人员编写比一般xUnit框架表达性更强，组织更好的代码。此外我们还力图减少依赖，这样你可以在Node.js上使用Jasmine，也可以在浏览器或移动程序中使用。”

本章将介绍：

- 初识Jasmine
- 组织测试用例
- 创建单元测试
- Jasmine的断言
- 测试替身（Test Double）
- 测试异步代码
- Jasmine插件
- 基于浏览器调试

4.1 初识Jasmine

4.1.1 获取Jasmine

访问Jasmine的发布网址https://github.com/jasmine/jasmine/releases并下载它的zip独

① Dio Synodinos. Virtual Panel: State of the Art in JavaScript Unit Testing[OL]. 2011. https://www.infoq.com/articles/javascript-unit-testing.

立发布包，如图4-1所示（本书编写时Jasmine的最新版本是2.5.2，jasmine-standalone-2.5.2.zip）。

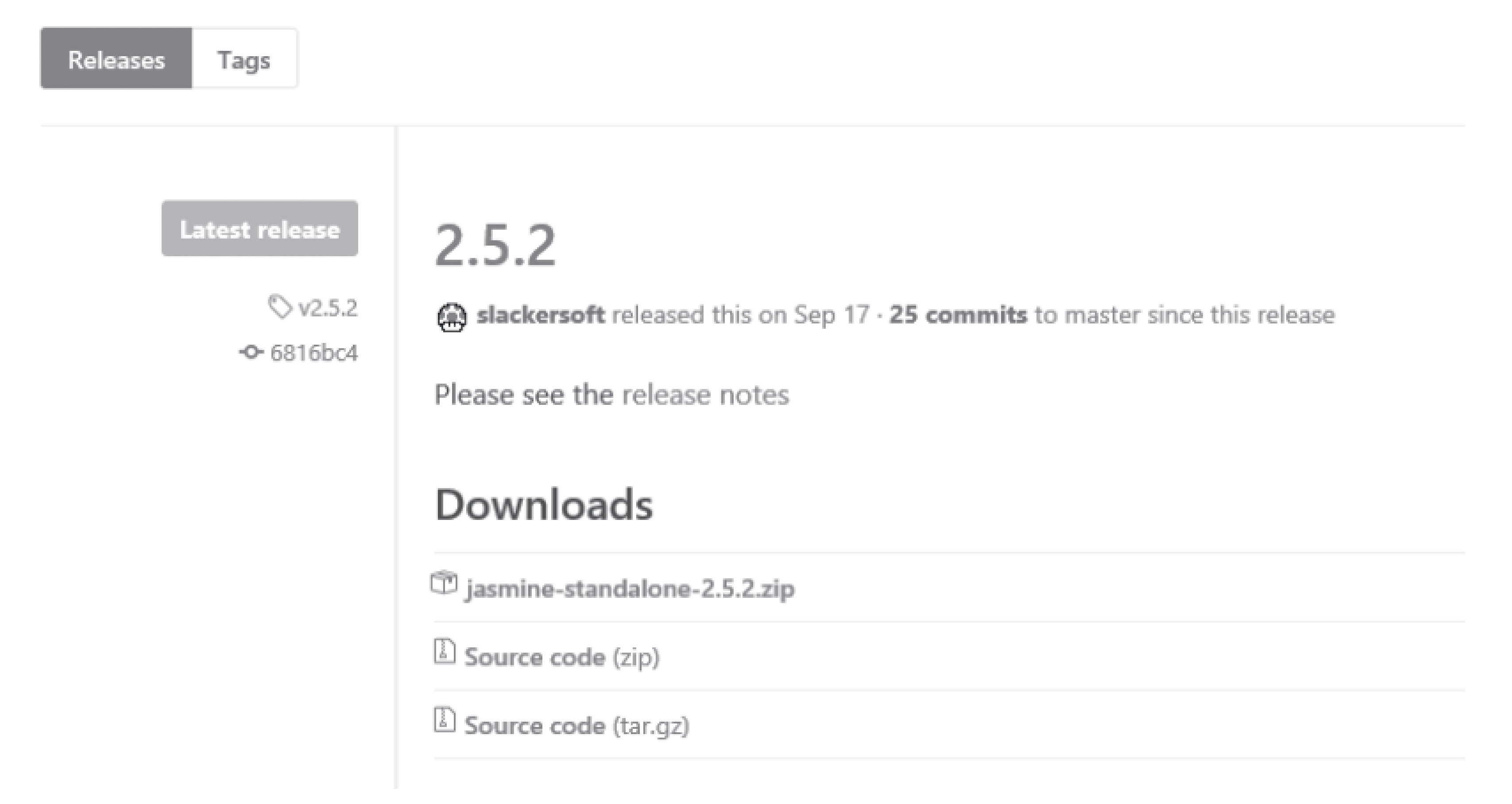


图4-1　下载Jasmine发布包

将zip文件解压，得到如图4-2所示的目录结构。

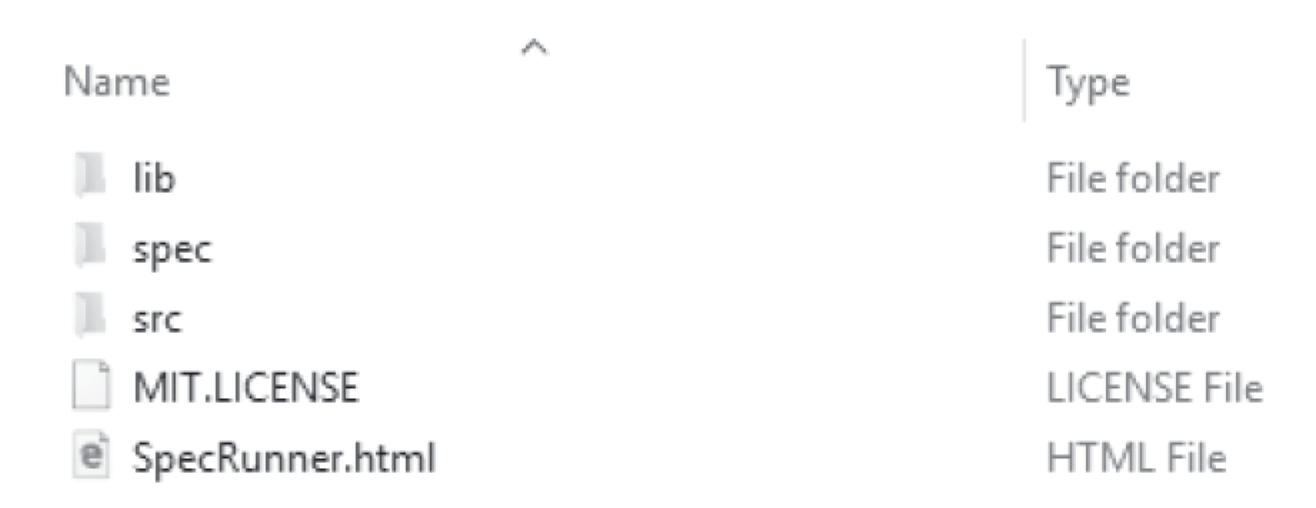


图4-2　Jasmine 独立发布包目录

双击SpecRunner.html，在浏览器里会看到如图4-3所示的结果。



图4-3　双击SpecRunner.html显示的结果

这是Jasmine发布包里附带的单元测试示例的运行结果。在解释什么是SpecRunner.html

文件之前，读者应先了解一下针对前端JavaScript应用进行单元测试的基本方法。

4.1.2 前端单元测试架构

通常，前端JavaScript单元测试需要准备以下文件，如表4-1所示。

表4-1 JavaScript单元测试所需文件

文 件 名	描 述
app.js	被测试的JavaScript应用（System under Test，缩写SUT）
testFramework.js	选用的测试框架
app.test.js	编写的测试用例
index.test.html	测试执行页面，引用了被测试的应用app.js、所有的依赖文件、测试框架testFramework.js和测试用例app.test.js

如图4-4所示，使用不同的浏览器加载index.test.html，执行测试用例，最后输出不同格式的测试结果。

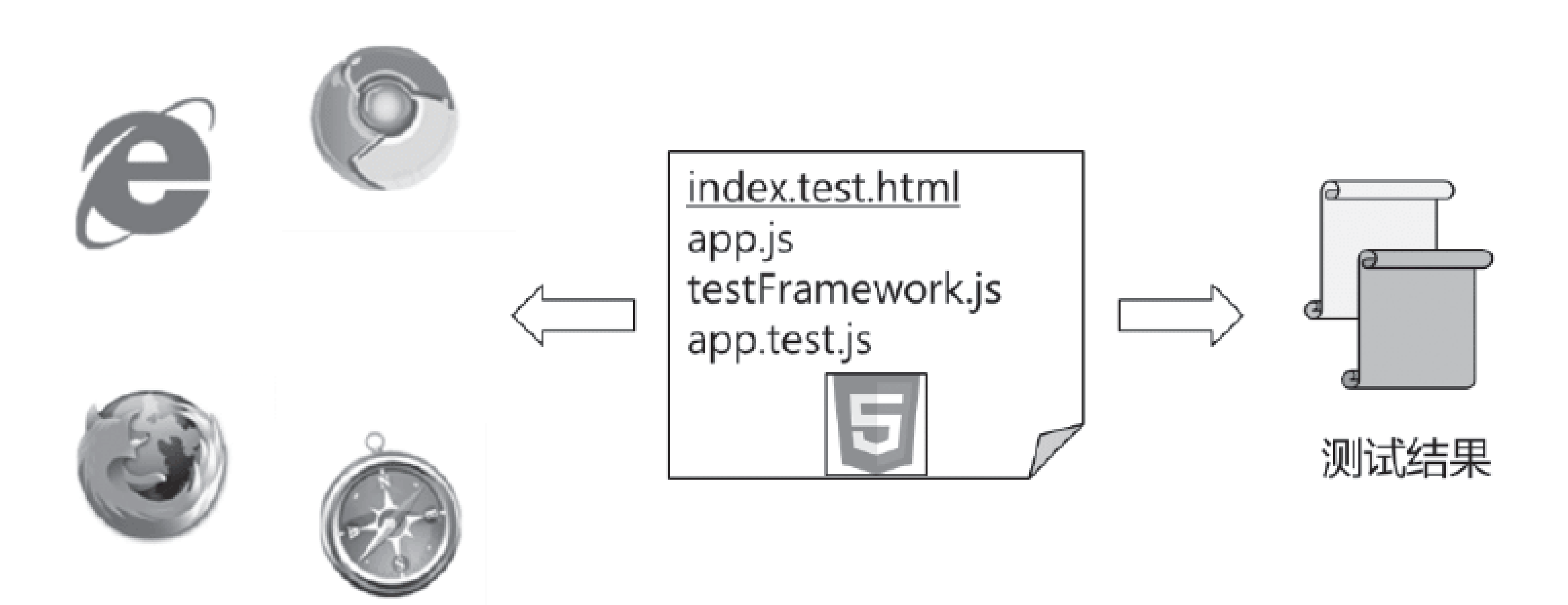


图4-4 JavaScript单元测试基本方法

4.1.3 Jasmine测试框架类库

在Jasmine的发布包里，src目录是被测试的JavaScript代码，示例包含了两个文件——Player.js和Song.js，如下所示。

```
-- src
  +-- Player.js
  +-- Song.js
```

在BDD里测试（test）描述了用户的需求，也是一个程序规格（spec）。如果编写的应用代码通过测试，那么意味着应用代码满足了用户需求，所以Jasmine里test被称为

spec。Jasmine的分发包里spec目录包含着两个测试相关的文件：PlayerSpec.js和SpecHelper.js，如下所示。

```
-- spec
  +-- PlayerSpec.js
  +-- SpecHelper.js
```

lib目录里是Jasmine的测试框架类库，类库文件的描述如表4-2所示。

表4-2 Jasmine测试框架类库

文 件 名	描 述
boot.js	用于初始化单元测试所需的执行环境类库
console.js	将单元测试结果输出到控制台的类库
jasmine.css	测试界面CSS样式
jasmine.js	执行单元测试的核心类库
jasmine_favicon.png	图标
jasmine-html.js	将单元测试结果输出为HTML格式的类库

最后，SpecRunner.html就是测试的执行页面（相当于图4-4的index.test.html）。使用任意文本编辑器打开这个文件，可以看到这个页面其实是一个容器，引用了测试需要的所有依赖文件，如图4-5所示。

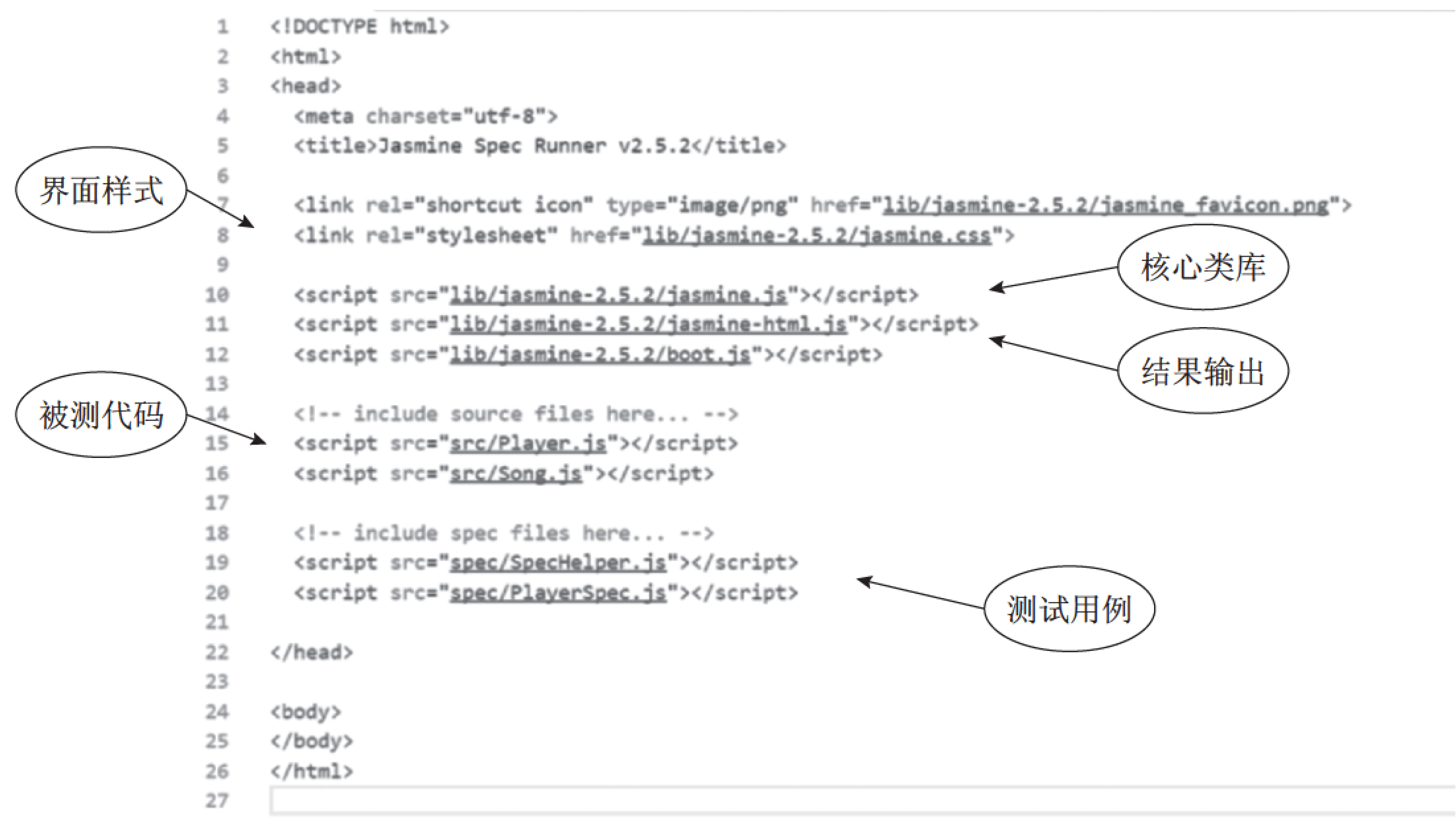


图4-5 SpecRunner.html

双击这个页面，会使用操作系统的默认浏览器加载这个页面并执行页面里引用的JavaScript代码，这些代码准备单元测试场景，执行测试用例，并将结果以HTML格式输出（jasmine-html.js），这就是图4-3所示的结果。也可以使用其他浏览器加载这个页面，这样测试用例就可以在不同的浏览器环境下执行，完成单元测试。

如果打开测试用例文件PlayerSpec.js，如图4-6所示，会看到有describe、beforeEach、it等函数，它们是什么意思呢？下一节详细介绍。

PlayerSpec.js ×

```
describe("Player", function() {
  var player;
  var song;

  beforeEach(function() {
    player = new Player();
    song = new Song();
  });

  it("should be able to play a Song", function() {
    player.play(song);
    expect(player.currentlyPlayingSong).toEqual(song);

    //demonstrates use of custom matcher
    expect(player).toBePlaying(song);
  });

```

图4-6　PlayerSpec.js

4.2　组织测试用例

Jasmine使用describe、it等全局函数组织测试用例。

4.2.1　describe

describe是Jasmine的全局函数，用于创建一个测试套件（test suite）。测试套件可以理解为一组测试用例（test case或spec）的集合。

describe函数接受两个参数（一个字符串和一个回调函数）。字符串参数是这个测试套件的名字或标题（通常描述被测试内容），回调函数是实现测试套件的代码块（称为describe块）。示例代码如下：

```
describe('Player', function() {
        /* tests */
});
```

describe函数可以嵌套使用（测试套件可以包含测试套件），例如：

```
describe('Player', function() {
    describe('when new', function() {
        /* tests */
    });
});
```

4.2.2 it

it也是Jasmine的全局函数，用来在describe块里创建一个测试用例（spec）。和describe一样，it接受两个参数（一个字符串和一个回调函数），字符串参数是测试用例的名字或标题，回调函数是实现测试用例的代码块（称为it块）。示例代码如下：

```
describe('Player', function() {
    it('should be able to play a Song', function() {
        /* code and assertions */
    });
    it('should be able to pause a Song', function() {
        /* code and assertions */
    });
});
```

describe和it函数的字符串参数很重要。describe创建的测试套件用来组织多个相关的测试用例。通过拼接测试套件的名字和测试用例的名字，可以完整描述一个测试场景，方便在大型项目中进行查找。如果描述得当的话，你的测试可以以自然语言的方式表达出来，形成文档。

describe块和it块都是JavaScript函数，所以JavaScript的作用域（scope）规则也适用于

此处。在describe块里声明的变量可以被它内部任何it块使用。

```
describe('Player', function() {
    var a;
    it('and so is a spec', function() {
        a = true;
    });
});
```

4.2.3 安装和拆卸

在真正执行测试代码之前，通常需要做大量的铺垫，例如准备一些测试数据，建立测试场景，这些为了成功测试而做的准备工作称为Test Fixture。测试完毕后需要释放运行测试占用的资源。这些铺垫工作占据的代码可能随着测试复杂度的增加而增加。为了避免在每个测试用例里重复这些代码，测试框架一般都会提供安装（Setup）和拆卸（Teardown）函数。

Jasmine提供了4个全局函数用于安装和拆卸，如表4-3所示。

表4-3 安装和拆卸函数

函数名称	描 述
beforeEach	在每一个测试用例（it块）执行之前都执行一遍beforeEach函数
afterEach	在每一个测试用例（it块）执行完成之后都执行一遍afterEach函数
beforeAll	在测试套件（describe块）中所有测试用例执行之前执行一遍beforeAll函数
afterAll	在测试套件（describe块）中所有测试用例执行完成之后执行一遍afterAll函数

这些函数接受一个回调函数作为参数，执行相关的安装代码和拆卸代码。例如：

```
describe('Player', function() {
    var player;
    beforeEach(function() {
        player = new Player();
    });
    afterEach(function() {
        /* cleanup code */
```

```
    });

    it('should be able to play a Song', function() {

      /* code and assertions */

    });

});
```

因为describe 可以嵌套，所以测试用例可以定义在任何一层describe块里，如图4-7所示。

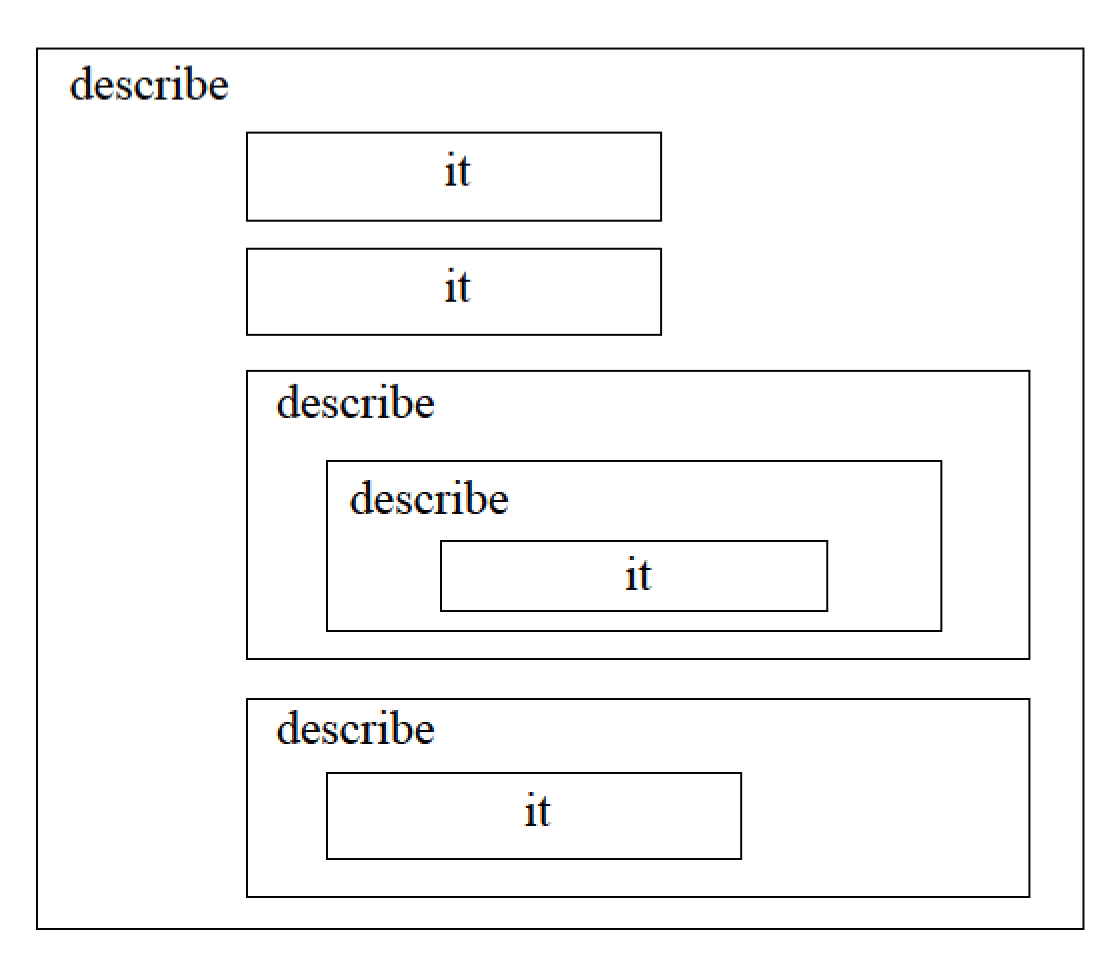


图4-7　嵌套的describe

理解安装和拆卸函数在嵌套describe情况下的执行顺序，有助于合理组织测试用例。

例如使用下面这个例子：

```
describe('Jasmine Execution Sequence', function () {

   beforeAll(function () {

      console.log('outer beforeAll');

   });

   beforeEach(function () {

      console.log('outer beforeEach');

   });

   it('spec 1', function () {

      console.log('spec 1');

   });
```

```
    console.log('statement 1');
    describe('inner', function () {
        beforeAll(function () {
            console.log('inner beforeAll');
        });
        afterAll(function () {
            console.log('inner afterAll');
        });
        console.log('statement 3');
        beforeEach(function () {
            console.log('inner beforeEach');
        });
        it('spec 3', function () {
            console.log('spec 3');
        });
        afterEach(function () {
            console.log('inner afterEach');
        });
    });
    it('spec 2', function () {
        console.log('spec 2');
    });
    console.log('statement 2');
    afterEach(function () {
        console.log('outer afterEach');
    });
    afterAll(function () {
        console.log('outer afterAll');
    });
});
```

输出结果如下：

```
statement 1
statement 3
statement 2
outer beforeAll
outer beforeEach
spec 1
outer afterEach
inner beforeAll
outer beforeEach
inner beforeEach
spec 3
inner afterEach
outer afterEach
inner afterAll
outer beforeEach
spec 2
outer afterEach
outer afterAll
```

以上示例有这样的输出结果是因为：

- Jasmine会先执行describe块的代码，然后再执行beforeAll、beforeEach和it函数。所以“statement 1”“statement 3”“statement 2”首先被输出。
- describe块的代码从上到下依次执行。尽管console.log('statement 2')在外层describe块里，但是它还是排在内层describe块的console.log('statement 3')后面执行。
- beforeAll会在它所在describe块的测试用例和beforeEach执行前执行，而且只执行一次。
- beforeEach会在它所在describe块和内层describe块里的测试用例执行前被执行。所以outer beforeEach会在外层的测试用例spec 1之前执行，也会在内层的测试用例

spec 3之前执行。而inner beforeEach只会在spec 3之前执行。

- 在每个测试用例执行前，Jasmine会从最外层的describe块开始，顺序执行每个beforeEach，直到这个测试用例所在的describe块为止。所以在执行测试用例spec 3之前，Jasmine先执行outer beforeEach，然后执行inner beforeEach。
- 测试用例会从上到下依次执行。虽然spec 2在外层，但是它还是在内层的测试用例spec 3后面执行。
- 测试用例执行完后，Jasmine会执行测试用例所在describe块的afterEach，然后依次执行外层的afterEach，直至最外层describe块。例如在spec 3测试用例完成后，inner beforeEach会先被执行，然后是outer afterEach。
- afterAll会在它所在describe块的测试用例和afterEach执行后执行，而且只执行一次。

4.2.4 禁用测试套件和挂起测试用例

有时候用户希望某些测试用例在单元测试时不被执行，因为这些测试用例还未完成，或者执行时间比较长。这时可以使用Jasmine提供的xdescribe函数来禁用测试套件，屏蔽被禁用的测试套件内的所有测试用例，使这些被禁用的测试用例不会出现在最终测试报告里。示例代码如下：

```
xdescribe('Player', function() {
        /* disabled tests */
});
```

测试用例也可以被挂起（pending）。和禁用不同，挂起的测试用例不会被执行，但是测试报告里会显示处于挂起状态的测试用例的名字。

有3种挂起测试用例的方式。

（1）使用xit函数，例如：

```
xit('can be declared "xit"', function () {
    /* code and assertions */
});
```

（2）使用it函数创建测试用例时，只提供第1个字符串名字，不提供第2个回调函数。例如：

```
it('can be declared with "it" but without a function');
```

（3）在it代码块的任意地方调用pending函数。例如：

```
it('can be declared by calling "pending" in the body', function () {
   /* code and assertions */
   pending('this is why it is pending');
});
```

测试报告会显示挂起的测试用例，如图4-8所示。

```
Jasmine 2.5.2
* * *
3 specs, 0 failures, 3 pending specs

  Pending specs
    can be declared "xit" PENDING WITH MESSAGE: Temporarily disabled with xit
    can be declared with "it" but without a function
    can be declared by calling "pending" in the body PENDING WITH MESSAGE: this is why it is pending
```

图4-8 挂起的测试用例

4.3 创建单元测试

前面解释了如何使用Jasmine组织测试用例，现在开始创建第1个单元测试项目。

4.3.1 准备测试场景

准备测试场景的步骤如下。

（1）创建一个目录jasmine-demo，在命令控制台里将当前目录切换到jasmine-demo。

（2）运行npm init命令生成package.json文件。（-y参数表示不进行交互，直接使用默认设置）

```
C:\jasmine-demo>npm init -y
```

（3）为了方便管理，这里使用npm install下载Jasmine的核心类库。Jasmine核心类库会被下载到node_modules子目录。

```
C:\jasmine-demo>npm install jasmine-core --save-dev
```

（4）创建src和spec子目录。src子目录用于存放被测代码，spec目录用于保存测试用例代码。

（5）在src和spec目录下分别创建Basic子目录，用来保存第一个单元测试示例的被测代码和测试用例。此时目录结构如下：

```
- jasmine-demo
  +-- node_modules
  | +-- jasmine-core
  +-- src
  | +-- Basic
  +-- spec
  | +-- Basic
```

（6）在jasmine-demo\src\Basic目录下创建Calc.js，该文件中是被测的JavaScript代码：

```
/* Calc.js */
var Calculator = function() {};


Calculator.prototype.add = function(a, b) {
     return a + b;
};
```

4.3.2 编写测试用例

编写单元测试用例一般遵循如下三个步骤的AAA模式。

（1）Arrange（准备）。设置测试场景，准备测试数据。

（2）Act（执行）。调用被测试代码。

（3）Assert（断言）。验证被测代码的行为是否与预期相同。

在jasmine-demo\spec\Basic目录下创建Calc_spec.js，添加如下测试代码：

```
/* Calc_spec.js */
describe('Calculator', function () {
   var calc;

   beforeAll(function () {
      //Arrange
      calc = new Calculator();
   });

   describe('Test Add', function () {
      it('add 1 and 3 should equal 4', function () {
         //Act
         var result = calc.add(1,3);
         //Assert
         expect(result).toBe(4);
      });
   });
});
```

以上测试代码里有一句断言：

```
expect(result).toBe(4);
```

单元测试验证测试结果通常采用断言（assertion）的形式，也就是测试某个功能的返回结果，是否与预期结果一致。如果与预期不一致，就表示测试失败。以上这个断言的意思是调用add(1,3)的结果应该是4。

所有的测试用例（it块）都应该包含有一句或多句的断言，它是编写测试用例的关键。

4.3.3 执行测试

为了执行测试用例，在jasmine-demo\spec\Basic目录下创建测试执行页面SpecRunner.html（参考独立分发包里的SpecRunner.html），内容如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Calculator Test</title>
    <link rel="shortcut icon" type="image/png" href="../../node_modules/jasmine-core/
images/jasmine_favicon.png">
    <link rel="stylesheet" href="../../node_modules/jasmine-core/lib/jasmine-core/
jasmine.css">
   <script src="../../node_modules/jasmine-core/lib/jasmine-core/jasmine.js"></script>
   <script src="../../node_modules/jasmine-core/lib/jasmine-core/jasmine-html.js">
</script>
   <script src="../../node_modules/jasmine-core/lib/jasmine-core/boot.js"></script>
   <!-- include source files here... →
   <script src="../../src/Basic/Calc.js"></script>
   <!-- include spec files here... →
   <script src="Calc_spec.js"></script>
</head>
<body>
</body>
</html>
```

测试执行页面引用了node_modules目录下的Jasmine核心类库，被测代码Calc.js以及测试用例代码Calc_spec.js。

双击该页面，默认浏览器执行测试代码并输出测试报告，如图4-9所示。



图4-9　Calculator单元测试报告

4.4　Jasmine的断言

Jasmine的断言很接近于自然语言。断言的写法分为两部分：

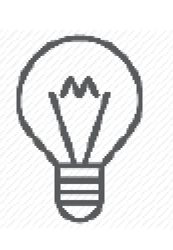
```
expect.matcher
```

第一部分expect函数接受一个参数，这个参数代表“实际值”，也就是被测代码的运行结果。

expect需要和第二部分matcher（匹配器）一起使用。匹配器接受“期望值”，将“期望值”与“实际值”进行布尔判断。Jasmine根据匹配器的结果决定测试用例是否通过测试。以前面Calc_spec.js里的断言为例：

```
expect(result).toBe(4);
```

result是“实际值”，toBe是Jasmine内建的一个匹配器，而4则是“期望值”。这个测试用例期望result的值是4。



如果匹配器要执行否定判断，可以在expect和匹配器间加上not，例如：

```
expect(false).not.toBe(true);
```

4.4.1　内置匹配器

Jasmine提供了丰富的内置匹配器。

1. toBe

该内置匹配器的语义是期望“实际值”和“期望值”严格相等（执行JavaScript"==="运算）。例如：

```
it('toBe', function () {
   var a = {};
   var b = a;
   var c = {};

   expect(a).toBe(b);
   expect(a).not.toBe(c);
});
```

如果想了解某个匹配器如何进行布尔判断，可以查阅jasmine.js文件中相应匹配器的源代码。例如toBe匹配器的代码如下：

```
getJasmineRequireObj().toBe = function() {
  function toBe() {
    return {
      compare: function(actual, expected) {
        return {
          pass: actual === expected
        };
      }
    };
  }
  return toBe;
}
```

2. toBeCloseTo

该内置匹配器的语义是期望“实际值”和“期望值”足够接近（不一定要相等）。

例如：

```
it('toBeCloseTo', function () {
    var a = 3.78,
        b = 3.76;
    expect(a).not.toBeCloseTo(b, 2);
    expect(a).toBeCloseTo(b, 1);
});
```

所谓“足够接近”，就是把两个数按照指定的精度进行四舍五入后比较是否相等。toBeCloseTo 的第二个参数用于指定精度。以上第一个断言里保留两位小数，所以“不接近”，第二个断言里只保留一位小数，所以“足够接近”。

3. toBeDefined

该内置匹配器的语义是期望“实际值”已定义。例如：

```
it('toBeDefined', function () {
    var a = {
        foo: 'foo'
    };
    expect(a.foo).toBeDefined();
    expect(a.bar).not.toBeDefined();
});
```

4. toBeFalsy

该内置匹配器用于实现布尔测试，使期望“实际值”为falsy。例如：

```
it('toBeFalsy', function () {
    var a, foo = 'foo';
    expect(a).toBeFalsy();
    expect(foo).not.toBeFalsy();
});
```



JavaScript里除了下列falsy值[①]，其余所有的值都是truthy，包括“0”、“false”、空的function、空的array和空的object。

- false
- null
- undefined
- 0
- NaN
- ‘’（空字符串）
- “”（空字符串）
- document.all

5. toBeTruthy

该内置匹配器用于实现布尔测试，使期望“实际值”为truthy，和toBeFalsy相反。例如：

```
it('toBeTruthy', function () {
   var a, foo = 'foo';
   expect(foo).toBeTruthy();
   expect(a).not.toBeTruthy();
});
```

6. toBeGreaterThan

该内置匹配器的语义是期望“实际值”大于“期望值”。例如：

```
it('toBeGreaterThan', function () {
   var a = 3.78,
      b = 3.76;
   expect(a).toBeGreaterThan(b);
   expect(b).not.toBeGreaterThan(a);
});
```

7. toBeGreaterThanOrEqual

该内置匹配器的语义是期望“实际值”大于或等于“期望值”。

① Mozilla Developer Network. Falsy[OL]. [2016]. https://developer.mozilla.org/en-US/docs/Glossary/Falsy.

8. toBeLessThanOrEqual

该内置匹配器的语义是期望“实际值”小于或等于“期望值”。

9. toBeLessThan

该内置匹配器的语义是期望“实际值”小于“期望值”。

10. toBeNaN

该内置匹配器的语义是期望“实际值”是NaN（Not-A-Number）[①]。例如：

```
it('toBeNaN', function () {
   expect(0 / 0).toBeNaN();
   expect(parseInt('foo')).toBeNaN();
   expect(5).not.toBeNaN();
});
```

11. toBeNull

该内置匹配器的语义是期望“实际值”是null。例如：

```
it('toBeNull', function () {
   var a = null;
   var foo = 'foo';
   expect(a).toBeNull();
   expect(foo).not.toBeNull();
});
```

12. toBeUndefined

该内置匹配器的语义是期望“实际值”未被定义，和toBeDefined相反。例如：

```
it('toBeUndefined', function () {
   var a = {
      foo: 'foo'
   };
   expect(a.foo).not.toBeUndefined();
```

① Mozilla Developer Network. NaN[OL]. 2015. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/NaN.

```
    expect(a.bar).toBeUndefined();
});
```

13. toContain

该内置匹配器的语义是期望“实际值”（数组或对象）包含“期望值”。例如：

```
describe('toContain', function () {
    it('finds an item in an Array', function () {
        var a = ['foo', 'bar', 'baz'];
        expect(a).toContain('bar');
        expect(a).not.toContain('quux');
    });
    it('finds a substring', function () {
        var a = 'foo bar baz';
        expect(a).toContain('bar');
        expect(a).not.toContain('quux');
    });
});
```

14. toEqual

该内置匹配器的语义是期望两个对象（“实际值”和“期望值”）相等。例如：

```
describe('toEqual', function () {
    it('works for simple literals and variables', function () {
        var a = 12;
        expect(a).toEqual(12);
    });
    it('works for objects', function () {
        var foo = {
            a: 12,
            b: 41
        };
```

```
        var bar = {
            a: 12,
            b: 41
        };
        expect(foo).toEqual(bar);
    });
});
```

查看Jasmine里toEqual函数的实现，会注意到toEqual函数接受cutomEqualityTesters参数，这个参数是自定义相等检验器。如果用户定义了自己的相等检验器，Jasmine会使用这些相等检验器对“实际值”和“期望值”进行相等比较。例如：

```
function toEqual(util, customEqualityTesters) {
    customEqualityTesters = customEqualityTesters || [];
    return {
      compare: function(actual, expected) {
      …
        result.pass = util.equals(actual,
          expected,
          customEqualityTesters);
      …
      }
    };
}
```

15. toMatch

该内置匹配器的语义是期望“实际值”能够匹配“期望值”，“期望值”可以是正则表达式，也可以是字符串。例如：

```
    it('toMatch is for regular expressions', function () {
        var message = "foo bar baz";
        expect(message).toMatch(/bar/);
```

```
    expect(message).toMatch("bar");
    expect(message).not.toMatch(/quux/)
});
```

16. toThrow

该内置匹配器的语义是期望“实际值”（函数）会抛出异常。例如：

```
it('toThrow', function () {
    var foo = function () {
        return 1 + 2;
    };
    var bar = function () {
        return a + 1;
    };
    expect(foo).not.toThrow();
    expect(bar).toThrow();
});
```

17. toThrowError

该内置匹配器的语义是期望“实际值”（函数）抛出“期望值”所指定的异常，异常信息可以是字符串、正则表达式、错误类型和错误信息。例如：

```
it('toThrowError is for testing a specific thrown exception', function() {
  var foo = function() {
    throw new TypeError("foo bar baz");
  };
  expect(foo).toThrowError("foo bar baz");
  expect(foo).toThrowError(/bar/);
  expect(foo).toThrowError(TypeError);
  expect(foo).toThrowError(TypeError, "foo bar baz");
});
```

4.4.2　自定义匹配器（Custom Matcher）

如果内置匹配器无法满足需求的话，可以编写自定义匹配器来封装匹配规则。

Jasmine使用工厂模式创建自定义匹配器。匹配器工厂是一个函数，函数名就是自定义匹配器的名称，也就是暴露给expect调用的名称。它接受两个参数：

- util，给匹配器使用的一组工具函数。
- customEqualityTesters，调用util.equals时所必需。

匹配器工厂返回一个对象，这个对象要包含名为compare的函数，即匹配器的对比函数。这个函数将实现自定义匹配规则。Jasmine在执行匹配时会调用compare函数。以自定义匹配器isBetween为例，示例代码如下：

```
var customMatchers = {
   isBetween: function(util, customEqualityTesters){
      return {
         compare:function(actual, min, max) {
            var result = {
               pass: false,
               message: 'Expected ' + actual + ' is not between '
                  + min + ' and ' + max
            };
            if(actual >= min && actual <= max){
               result.pass = true;
                result.message = 'Expected ' + actual + ' is between '
                  + min + ' and ' + max;;
            }
            return result;
         }
      };
   }
};
```

isBetween是匹配器工厂，也是要实现的自定义匹配器的名称。它返回一个对象，这个对象里的compare函数接受的第一个参数是actual，代表“实际值”。后面的参数是可选参数，代表“期望值”。在上面的示例里，期望“实际值”在“期望值”min和max之间。

compare函数必须返回一个结果对象。该对象必须包含一个布尔值类型的pass 成员属性，告诉Jasmine匹配结果。在上面的示例里，一旦“实际值”在“期望值”min和max之间，则将pass属性设为true。

如果匹配失败，但是在返回的result对象中包含了message成员属性的话，Jasmine会使用message的值作为错误提示。

通常在beforeEach里使用jasmine.addMatchers函数注册自定义匹配器。该函数接受一个对象参数，这个对象可以包含多个匹配器工厂（在上面的示例里customerMatchers就是这样一个对象，isBetween是其中的一个字段）。例如，使用下面的代码：

```
beforeEach(function() {
  jasmine.addMatchers(customMatchers);
});
```

就可以在测试用例里使用isBetween自定义匹配器了。

```
it('against isBetween', function(){
    expect(8).isBetween(4, 10);
});
```

如果用户的自定义匹配器需要控制.not的行为（不是简单的布尔值取反），那么匹配器工厂返回的对象里除了compare函数，还需要包含另外一个函数negativeCompare。这样，Jasmine使用.not的时候会执行negativeCompare函数。

4.4.3 自定义相等检验器（Custom Equality Tester）

在Jasmine里用户可以使用toEqual内置匹配器判断两个对象是否相等。如果内置匹配器的默认规则无法满足需求的话，可以创建自定义相等检验器来使用自己的相等规则。

例如，对于以下示例里的Duck类型，用户希望两个Duck对象相等的条件是canSwim、canWalk和color相等，而canFly或其他属性不属于判断条件。这种情况就需要使用自定义相等检验器。

```
var Duck = function () {
   this.canSwim = true;
   this.canWalk = true;
   this.color = 'white';
   this.canFly = false;
};
Duck.prototype.swim = function(){
   if(this.canSwim){
      //code
   }
};
Duck.prototype.walk = function(){
   if(this.canWalk){
      //code
   }
};
```

自定义相等检验器是一个函数，它接受两个参数，也就是要进行比较的对象。自定义相等检验器根据相应条件对这两个对象进行比较，结果返回true或false值。如果自定义相等检验器无法进行判断，则返回undefined。以下示例里，仅对两个对象的canSwim、canWalk和color属性进行比较。

```
describe('Custom Equality Test', function () {
   var duckCustomEquality = function (first, second) {
      return first.canSwim == second.canSwim
               && first.canWalk == second.canWalk
               && first.color == second.color;
   };
   …
});
```

使用自定义相等检验器前需要注册。注册自定义相等检验器通常通过在beforeEach函数里调用jasmine.addCustomEqualityTester函数实现。示例代码如下：

```
beforeEach(function () {
    jasmine.addCustomEqualityTester(duckCustomEquality);
});
```

注册后就可以比较两个Duck对象了：

```
it('against Duck class', function () {
    var d = {
        canSwim: true,
        canWalk: true,
        canClimb: true,
        color: 'white',
        canFly: true
    };
    var duck = new Duck();
    expect(d).toEqual(duck);
    d.canSwim = false;
    expect(d).not.toEqual(duck);
});
```

4.4.4 非对称相等检验器（Asymmetric Equality Tester）

自定义相等检验器用来检验两个对象是否相等。如果只需要检验某一个对象是否满足特定条件，而不是两个对象严格相等时，我们可以自定义一个非对称相等检验器。它作为一个“期望值”对象出现，必须包含一个名为asymmetricMatch的方法。Jasmine在使用toEqual比较“实际值”和非对称相等检验器时会调用这个asymmetricMatch方法。

```
describe('Asymmetry Match', function() {
  var tester = {
```

```
    asymmetricMatch: function(actual) {
      var secondValue = actual.split(',')[1];
      return secondValue === 'bar';
    }
  };
  it('dives in deep', function() {
    expect('foo,bar,baz,quux').toEqual(tester);
  });
});
```

以上示例创建了非对称相等检验器tester，它的asymmetricMatch方法期望输入的“实际值”字符串以逗号分隔后第二个内容为bar。

4.4.5　辅助匹配函数

针对一些常用情况，Jasmine提供辅助函数帮助开发人员进行匹配。这些辅助函数会返回非对称相等检验器（具有asymmetricMatch方法），封装了需要匹配的“期望值”和特殊匹配规则。

1. jasmine.any

通常传给匹配器的“期望值”是一个数字、字符串或对象等具体实例，然后将之和“实际值”做比较。有时候用户希望判断“实际值”是否是某种类型，而不是具体实例，这时可以用 jasmine.any 封装匹配类型，例如：

```
describe('jasmine.any', function () {
    it('matches any value', function () {
        expect({}).toEqual(jasmine.any(Object));
        expect(12).toEqual(jasmine.any(Number));
    });
});
```

jasmine.any函数接受构造函数或类名作为参数。这个构造函数或类名被jasmine.any封装成“期望值”，期望“实际值”所具备的类型。以上示例里“实际值”{}是一个

Object，12是一个Number，所以测试通过。

2. jasmine.anything

jasmine.anything代表任何存在的值。只要“实际值”不是null或undefined，测试就通过。例如：

```
describe('jasmine.anything', function () {
    it('matches anything', function () {
        expect(1).toEqual(jasmine.anything());
    });
});
```

3. jasmine.objectContaining

jasmine.objectContaining用来判断“实际值”对象是否包含某个键值对。例如：

```
describe('jasmine.objectContaining', function () {
   var foo;
   beforeEach(function () {
      foo = {
         a: 1,
         b: 2,
         bar: 'baz'
      };
   });
   it('matches objects with the expect key/value pairs', function () {
      expect(foo).toEqual(jasmine.objectContaining({
         bar: 'baz'
      }));
      expect(foo).not.toEqual(jasmine.objectContaining({
         c: 37
      }));
   });
});
```

4. jasmine.arrayContaining

jasmine.arrayContaining用来判断输入的“期望值”是否是“实际值”数组的一部分。例如：

```
describe('jasmine.arrayContaining', function () {
    var foo;
    beforeEach(function () {
        foo = [1, 2, 3, 4];
    });
    it('matches arrays with some of the values', function () {
        expect(foo).toEqual(jasmine.arrayContaining([3, 1]));
        expect(foo).not.toEqual(jasmine.arrayContaining([6]));
    });
});
```

5. jasmine.stringMatching

jasmine.stringMatching用来判断输入的“期望值”是否是“实际值”字符串的一部分。如果“期望值”是正则表达式，那么就会判断“实际值”是否满足正则表达式。它也可以用来判断对象内的字符串。示例代码如下：

```
describe('jasmine.stringMatching', function () {
    it('matches as a regexp', function () {
        expect({ foo: 'bar' }).toEqual({
            foo: jasmine.stringMatching(/^bar$/)
        });
        expect({ foo: 'foobarbaz' }).toEqual({
            foo: jasmine.stringMatching('bar')
        });
    });
});
```

4.5 测试替身（Test Double）

开发人员编写的测试用例会设置有用户场景，执行应用的某个特定功能单元，然后验证功能单元的返回结果。如果被测的功能单元不依赖其他单元，那么测试会相对简单。但是在实际环境下，一般被测单元总会依赖其他的功能单元：

- 被测单元需要输入数据，它的运行会受到输入数据的影响。这些输入数据可能是测试用例以回调函数等形式提供给被测的功能单元，也有可能是被测单元调用它所依赖的第三方功能单元的API，然后以依赖单元的返回值作为输入数据，例如通过访问数据库获得数据。
- 被测单元需要输出结果。输出的结果可能是以函数返回值的形式返回给宿主程序，也可能功能单元本身不直接返回数据，而是调用一系列第三方功能单元的API，这些调用依赖单元的行为也是一种输出结果。

当被测单元依赖第三方单元时，单元测试就变得复杂起来。虽然可以将功能单元和它的依赖单元一起进行测试，但是这些依赖单元可能在测试环境中难以建立，或者难以返回测试所需要的数据。此外，单元测试需要尽可能快速地运行，若被测单元调用依赖的第三方子系统（比如数据库）则可能会花费比较长的时间。

另一种方案就是将被测的功能单元和依赖单元隔离，创建一些比较简单但是行为和实际依赖单元类似的假单元来代替真实的依赖单元，以降低测试的复杂性和提高测试的可行性。

Gerard Meszaros借鉴电影替身（Double）的概念，将这些假单元称为测试替身[①]（Test Double）。

4.5.1 测试替身的类型

测试替身主要提供两项功能：为被测单元提供输入数据和记录被测单元的输出结果。

当依赖单元提供输入数据时，单元测试需要控制依赖单元的行为，让依赖单元提供不同类型的数据来测试功能单元的各种行为。如果用测试替身替换依赖单元，则测试替身一般会模拟提供以下种类的测试数据：

① Gerard Meszaros. xUnit Test Patterns: Refactoring Test Code[M]. Addison-Wesley, 2007.

- 方法/函数的返回值
- 可更新参数的值
- 可以抛出的异常

执行单元测试后需要验证测试结果，有时候被测单元本身不返回测试结果，而是调用依赖单元（例如日志记录模块）间接输出结果，这时是无法通过函数返回值来验证被测单元是否执行成功的，而需要检查日志记录模块是否记录了相应的操作，来间接验证测试结果。如果用测试替身替换依赖单元，那么测试替身需要有记录调用行为的功能。

根据使用形式的不同，测试替身有以下类型。

1. Dummy Objects

被测单元的函数/方法可能会接受一些参数，然后将传入的参数对象存在实例变量里供以后使用。有时候这些参数对象不会在当前的单元测试过程中被使用，所以它们不会影响被测函数/方法的行为。

因为这些参数对象不会被使用，所以在单元测试中创建这些实际的参数对象是没有必要的，而只需要简单并且没有依赖的“假”对象。这样的“假”对象就是Dummy Object。

最简单的Dummy Object是null（nil，nothing），但有时候被测函数/方法要求输入的参数是not-null，这种情况下只能被迫创建一个真实对象。对于动态类型语言（例如JavaScript），可以使用String或Object；对于静态类型语言（例如C#、Java），必须确保Dummy Object和对应的参数对象类型相匹配。

2. Test Stubs

被测单元在执行时通常需要调用一些第三方的依赖单元。调用依赖单元的结果会影响到被测单元的行为。换一个角度讲，调用依赖单元的结果成为了被测单元的输入数据。在单元测试中，利用“假”的对象代替实际的依赖单元，这样 “假”对象返回的各种预设结果会影响被测单元的行为，确保被测单元内的代码都被测试到。这种 “假”对象称为Test Stub，如图4-10所示。

Test Stub一般会实现依赖单元的接口（interface）预先设置返回结果。在准备测试场景时，Test Stub用来取代实际的依赖单元，这样测试执行时被测单元调用Test Stub，Test Stub返回预先设置的结果给被测单元，两者的交互完全在被测单元内部，对测试用例透明。当被测单元执行完成后，测试用例会验证被测单元的执行结果。

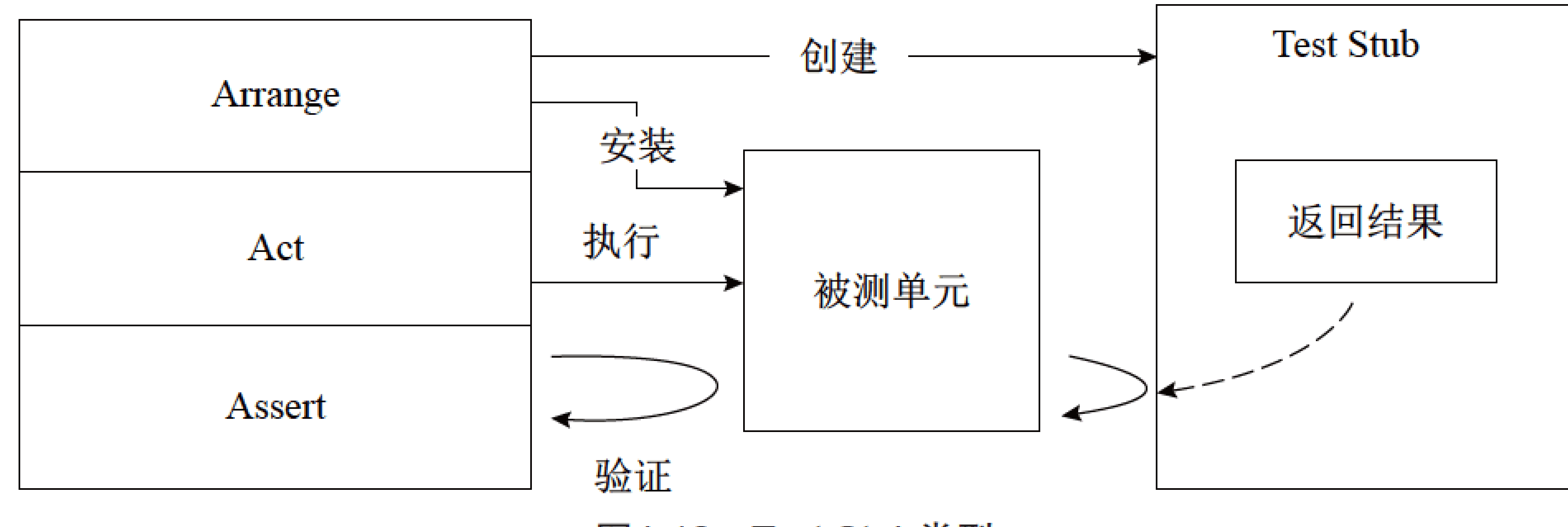


图4-10　Test Stub类型

3. Test Spies

有时被测单元本身不返回测试结果，但它会调用依赖单元间接输出执行结果，例如日志记录模块。为了验证被测单元是否按预期执行，可利用“假”对象来代替实际的依赖单元。这个“假”对象会记录并保存所有对它的调用信息。这种假对象称为Test Spy，如图4-11所示。

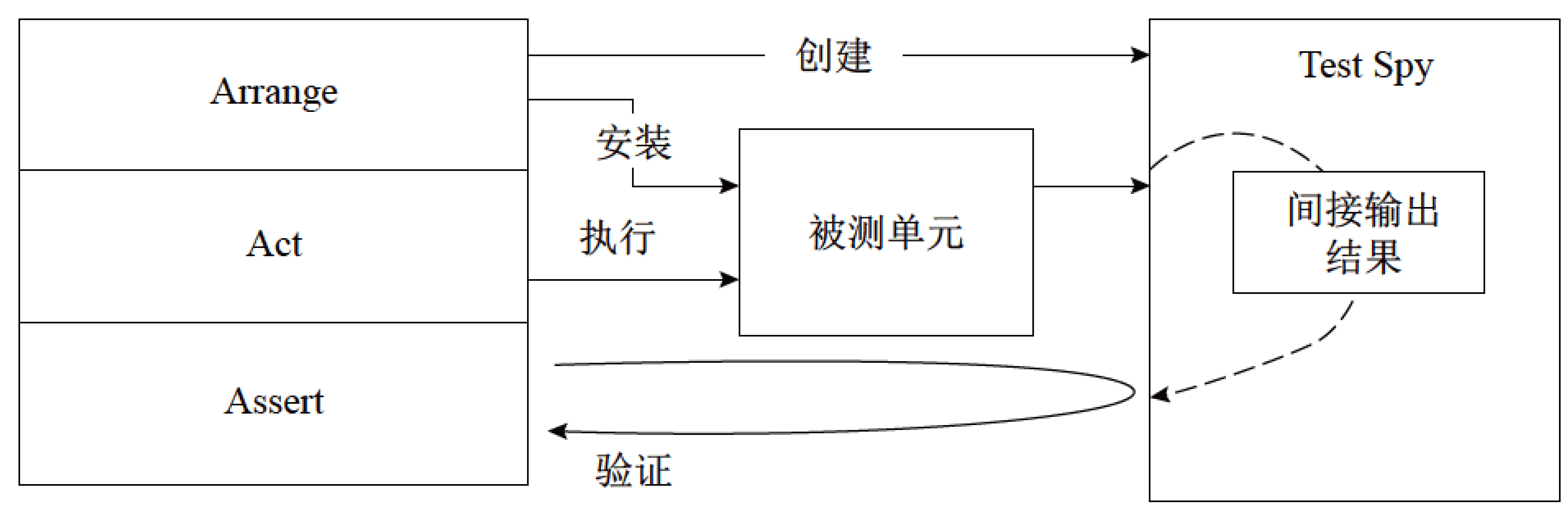


图4-11　Test Spy类型

当被测单元执行完成后，测试用例会检查Test Spy中保存的信息来验证被测单元是否按预期执行。

4. Mock Objects

Mock Object和Test Spy类似，也是代替实际的依赖单元，记录并保存被测单元对它的所有调用信息。和Test Spy不同的是，Mock Object除了记录信息以外，它会将被测单元对它的调用和测试用例预先设置的期望行为进行比较，一旦发现被测单元的行为偏离预期，就立即使测试失败。换句话说，Mock Object在Test Spy的基础上加入了验证的功能，如图4-12所示。

由于Mock Object封装了测试验证的逻辑，所以Mock Object可以被不同测试用例所重用。在使用Mock Object的情况下，测试用例本身理论上不需要验证代码，它完全信任Mock Object的验证结果。Mock Object一旦发现被测单位行为异常，可以立即使测试失

败，这样可以快速有效地定位错误发生的位置。与之相反，Test Spy没有验证功能，只能依赖测试用例在被测单元执行完成后再进行判断，所以Test Spy必须记录更详细的诊断信息才能定位错误位置。

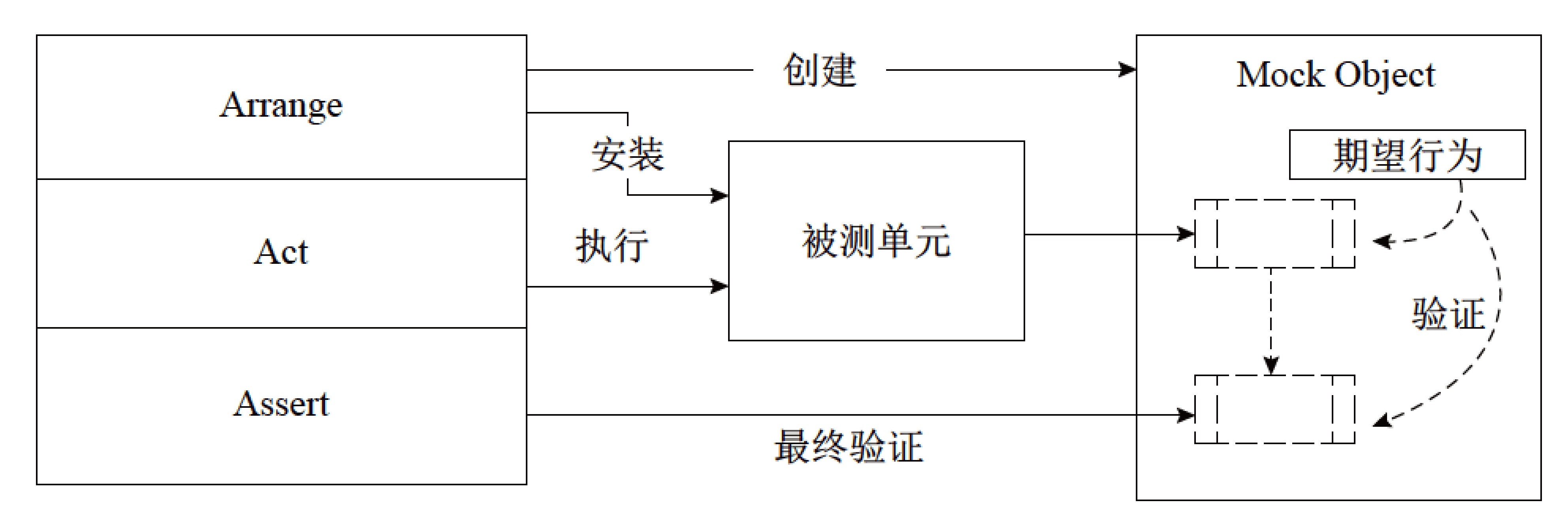


图4-12　Mock Object类型

5. Fake Objects

有时在测试环境中建立依赖单元比较困难，或者调用依赖单元要花很长时间，就要用到Fake Objects。Fake Object拥有几乎和实际依赖单元一样的功能，用它来替换实际依赖单元，被测单元仍然能够正常工作，如图4-13所示。例如使用一个内存数据库取代实际的数据库就是一个常见的Fake Object使用场景。

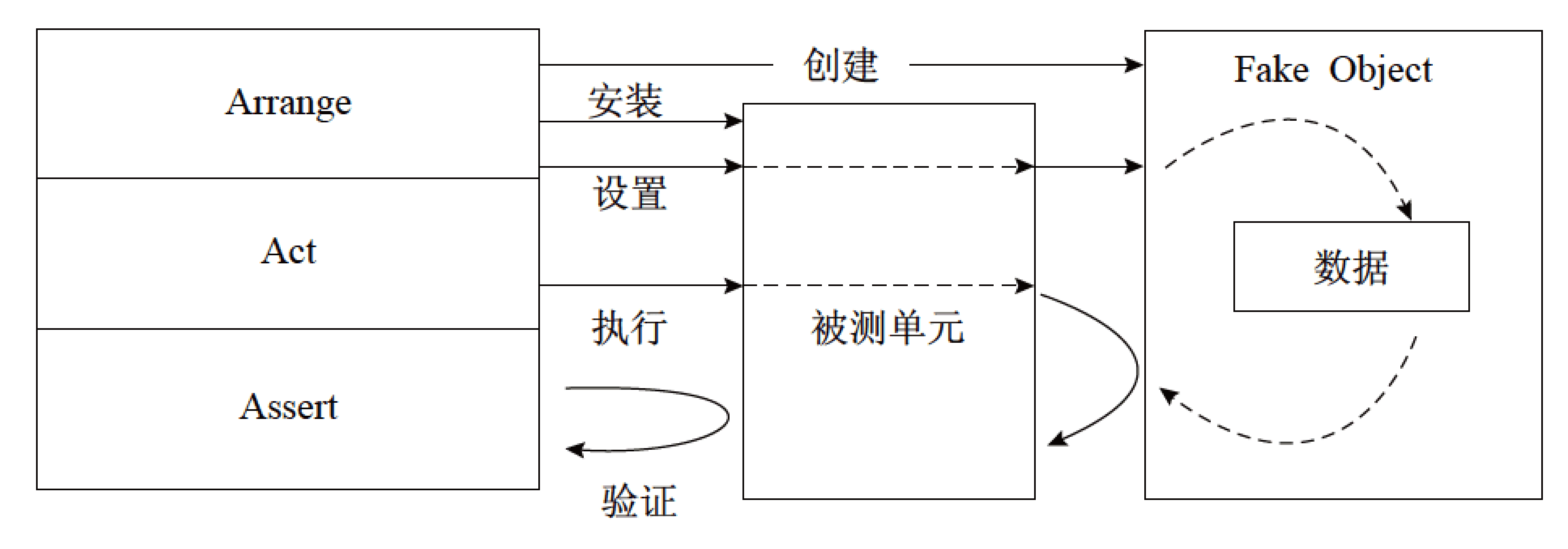


图4-13　Fake Object类型

Fake Object和Test Stub、Test Spy以及Mock Object的区别在于它只是为了减少对外部环境的依赖，是一种替代实现，并不提供控制输入输出和验证等功能。

4.5.2　使用Jasmine Spies

了解了什么是测试替身，那么如何创建测试替身呢？虽然用户可以根据测试需求进行手工创建，但是效率很低，因此通常会使用一些流行的JavaScript测试替身库（例如Sinon.JS）来实现。Jasmine可以使用Sinon.JS，但Jasmine其实有内建的测试替身库。

在Jasmine里，各种测试替身统称为Spy。开发人员使用Jasmine Spies来替代真实的函数/方法或者对象，并且跟踪它们的调用记录和传入的所有参数。Jasmine Spies只存在于describe块或者it块中，在每个测试用例使用结束后被销毁。

1. 使用spy跟踪函数的调用

假设类CarAssemble有3个成员函数addWheel、addEngine和assemble，其中，assemble函数会调用addWheel和addEngine（assemble依赖addWheel和addEngine），但是assemble函数本身不返回结果，有如下代码：

```
var CarAssemble = function () {
   this.wheel = 0;
   this.engine = null;
};
CarAssemble.prototype.addWheel = function(){
   this.wheel += 1;
};
CarAssemble.prototype.addEngine = function(engineName){
   this.engine = engineName;
};
CarAssemble.prototype.assemble = function(){
   this.addWheel();
   this.addWheel();
   this.addWheel();
   this.addWheel();
   this.addEngine('V8');
};
```

因为assemble函数没有返回结果，为了验证它是否成功执行，需要使用spy来跟踪addWheel和addEngine的调用。以下是测试用例代码：

```
describe('Spies Test', function () {
   it('for spyOn against CarAssemble function', function () {
      var fake = new CarAssemble();
```

```
        // Replace addWheel function with a spy
        spyOn(fake, 'addWheel');
        // Replace addEngine function with another spy
        spyOn(fake, 'addEngine');
        fake.assemble();
        expect(fake.addWheel).toHaveBeenCalled();
        expect(fake.addWheel).toHaveBeenCalledTimes(4);
        expect(fake.addEngine).toHaveBeenCalledWith('V8');
        expect(fake.addWheel.calls.any()).toEqual(true);
        expect(fake.addWheel.calls.count()).toEqual(4);
    });
});
```

以上测试用例中，为了测试assemble函数，首先创建了一个CarAssemble对象，然后调用spyOn函数创建spy代替addWheel和addEngine这两个实际函数。此时fake.addWheel和fake.addEngine已经不是实际函数而是spy了。

spyOn 是Jasmine的一个全局函数，用来创建一个spy并且代替一个已存在的函数。其用法如下：

```
// assumes obj.method is an existing function
spyOn(obj, 'method');
```

接下来执行被测代码fake.assemble并验证spy是否如预期被调用。Jasmine提供了spy相关的匹配器，如表4-4所示。

表4-4 spy相关的匹配器

匹　配　器	描　　述
toHaveBeenCalled	测试spy是否被调用过。如果被调用，返回true
toHaveBeenCalledTimes	测试spy是否被调用过指定的次数。示例代码里期望fake.addWheel被调用4次
toHaveBeenCalledWith	测试spy被调用时的参数列表。若匹配则返回true。示例代码里期望fake.addEngine被调用时传入的参数是‘V8’

如果需要进行否定判断，例如期望spy被调用时不会传入某些参数，可以在expect和匹配器之间加not。例如：

```
expect(fake.addEngine).not.toHaveBeenCalledWith('V6');
```

除了以上匹配器，访问spy的calls属性还可以获得对spy的调用信息，如表4-5所示。

表4-5 spy的calls属性

calls属性	描 述
.calls.any()	记录spy是否被调用过，如果没有，返回false，否则，返回true
.calls.count()	返回spy被调用过的次数
.calls.argsFor(index)	返回spy被第index次调用时的参数（index从0开始计算）
.calls.allArgs()	返回spy所有被调用的参数
.calls.all()	返回spy所有被调用的this 上下文和参数
.calls.mostRecent()	返回spy最近1次被调用的this上下文和参数
.calls.first()	返回spy第1次被调用的this上下文和参数
.calls.reset()	清除spy的所有调用记录

2. 使用spy调用实际函数

spyOn函数创建的spy只能记录被调用的信息，而不会修改任何数据，也不影响其他函数或对象的状态。有时除了需要跟踪函数的调用，用户还希望在测试时能执行实际的函数，并更新被测单元的数据。此时只要在spyOn创建的spy后面链式调用and.callThrough，就可以让spy在被调用时，除了记录调用信息，还继续调用实际函数。

```
it('for spyOn when configured to call through', function () {
   var fake = new CarAssemble();
   spyOn(fake, 'addEngine').and.callThrough();
   fake.assemble();
   expect(fake.addEngine).toHaveBeenCalled();
   expect(fake.engine).toBe('V8');
});
```

3. 使用spy控制函数的返回结果

单元测试中，有时希望利用“假”对象返回各种可能的结果来影响被测单元的行为，这时可以在spyOn函数创建的spy后面链式调用and.returnValue，指定返回结果。这样每次调用spy都可以得到这个指定的返回值。

```
it('for spyOn when configured to fake a return value', function() {
```

```
    var engineSupplier = {
        getEngine: function() {
            return 'V8';
        }
    };
    spyOn(engineSupplier, 'getEngine').and.returnValue('V6');
    var val = engineSupplier.getEngine();
    expect(val).toBe('V6');
    expect(val).not.toBe('V8');
});
```

如果要设定一个结果列表，可以在spy后面链式调用and.returnValues，每次调用spy，就会依次从这个结果列表返回一个值。如果所有的值都被返回了，那么Jamsine会返回undefined。例如：

```
it('for spyOn when configured to fake a series of return values', function() {
    var engineSupplier = {
        getEngine: function() {
            return 'V8';
        }
    };
    spyOn(engineSupplier, 'getEngine').and.returnValues('V6', 'V4');
    expect(engineSupplier.getEngine()).toBe('V6');
    expect(engineSupplier.getEngine()).toBe('V4');
    expect(engineSupplier.getEngine()).toBeUndefined();
});
```

如果在spy后面链式调用and.throwError，那么任何对该spy的调用都会抛出指定的错误。示例代码如下：

```
it('for spyOn when configured to throw an error', function () {
    var engineSupplier = {
```

```
        getEngine: function () {
            return 'V8';
        }
    };
    spyOn(engineSupplier, 'getEngine').and.throwError('broken engine');
    expect(function () {
        engineSupplier.getEngine();
    }).toThrowError('broken engine');
});
```

链式调用and.callThrough、and.returnValue等其实是为创建的spy增加了新功能，用户可以在任何时候调用.and.stub将spy恢复到原始状态（去除这些新功能，只能记录调用信息）。示例代码如下：

```
it('can fake a value and then stub in the same spec', function () {
    var engineSupplier = {
        getEngine: function () {
            return 'V8';
        }
    };
    spyOn(engineSupplier, 'getEngine').and.returnValue('V6');
    expect(engineSupplier.getEngine()).toBe('V6');
    engineSupplier.getEngine.and.stub();
    expect(engineSupplier.getEngine()).toBeUndefined();
});
```

4. 使用spy将实际函数替换成新函数

有时候需要在测试时使用一个全新的函数实现替换实际的函数，此时可以在Jasmine中spyOn后面的链式中调用and.callFake，以指定一个新函数。例如：

```
it('for spyOn when configured with an alternate implementation', function() {
    var engineSupplier = {
```

```
        getEngine: function() {
            return 'V8';
        }
    };
    var fakeEngine = function() {
        return 'Faked Engine';
    };
    spyOn(engineSupplier, 'getEngine').and.callFake(fakeEngine);
    expect(engineSupplier.getEngine()).toBe('Faked Engine');
});
```

5. 创建新的spy函数

前面介绍了在单元测试中创建spy替换实际已存在的函数（例如addWheel）。有时在单元测试中技术人员只需要一个“空”函数，这时可使用jasmine.createSpy来创建一个全新的spy函数（不用替换实际函数）。和其他spy一样，这个新的spy函数可以跟踪函数的调用和传入的参数，但是本身没有实现代码。示例代码如下：

```
it('for a spy, when created manually', function() {
    var engine = jasmine.createSpy('Named Engine');
    engine('Faked Engine');
    expect(engine).toHaveBeenCalled();
    expect(engine).toHaveBeenCalledWith('Faked Engine');
});
```

和spyOn一样，在createSpy后面也可以链式调用and.ReturnValue、and.callFake等函数。例如：

```
    jasmine.createSpy('V6 Engine').and.returnValue('V6');
    jasmine.createSpy('V8 Engine').and.callFake(function() {
        return 'V8';
    });
```

6.创建spy对象

使用jasmine.createSpyObj函数可以创建一个spy对象。示例代码如下：

```
it('for multiple spies, when created manually', function() {
    var engine = jasmine.createSpyObj('Named Engine', ['start', 'stop']);
    engine.start();
    engine.stop();
    expect(engine.start).toBeDefined();
    expect(engine.stop).toBeDefined();
    expect(engine.start).toHaveBeenCalled();
    expect(engine.start).toHaveBeenCalled();
});
```

jasmine.createSpyObj函数的第1个参数是对象名称，第2个参数是字符串数组，数组内每个字符串代表spy对象的一个属性，都是spy函数。在上面的示例中，start和stop都是engine对象的成员函数，作为spy函数使用。

4.6 测试异步代码

虽然JavaScript的执行环境是单线程的，但是JavaScript支持异步模式，有很多通过回调函数来执行的异步调用，例如setTimeout或setInterval。JavaScript单元测试需要考虑测试以下3种异步代码：

- 包含调用setTimeout或setInterval的代码。
- 需要花费一点时间才能显示的界面效果，例如网页元素的淡进淡出。
- Ajax调用。(在下一节将介绍使用Jasmine插件进行测试的方法)

为了演示测试异步代码，在jasmine-demo\src\Async目录下创建Engine.js。以下是被测的JavaScript代码。

```
/* Engine.js */
var Engine = function(displayElement) {
```

```
    this.$el = $(displayElement);
};
Engine.prototype.start = function(cb) {
    this.$el.fadeOut(1000, cb);
};
```

以上代码中，start函数里使用jQuery的fadeOut函数实现指定页面元素的淡出效果，时长为1秒（1000毫秒），元素淡出后会执行回调函数cb。

在jasmine-demo\spec\Async目录下创建Engine_spec.js文件，添加以下测试代码：

```
/* Engine_spec.js */
describe('Engine', function () {
   describe('UI Tests', function () {
      var engine, el;
      beforeEach(function () {
         el=$('#fade-div');
         engine = new Engine(el);
         engine.start();
      });
      it('should work with a visual effect', function () {
         expect(el.css("display")).toBe("none");
      });
   });
});
```

以上示例使用了jQuery获取页面中的fade_div元素，在测试用例中调用engine.start，然后验证fade_div元素是否消失。

因为使用了jQuery，所以SpecRunner.html需要引用jQuery库，同时也预先准备了fade_div元素，供测试使用。该.html文件内容如下：

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
…
    <script src="../../node_modules/jquery/dist/jquery.min.js"></script>
    <!-- include source files here... →
    <script src="../../src/Async/Engine.js"></script>
    <!-- include spec files here... →
    <script src="Engine_spec.js"></script>
</head>
<body>
    <div id="fade-div">some content</div>
</body>
</html>
```

双击SpecRunner.html文件，默认浏览器执行测试代码并且报告测试失败，如图4-14所示。

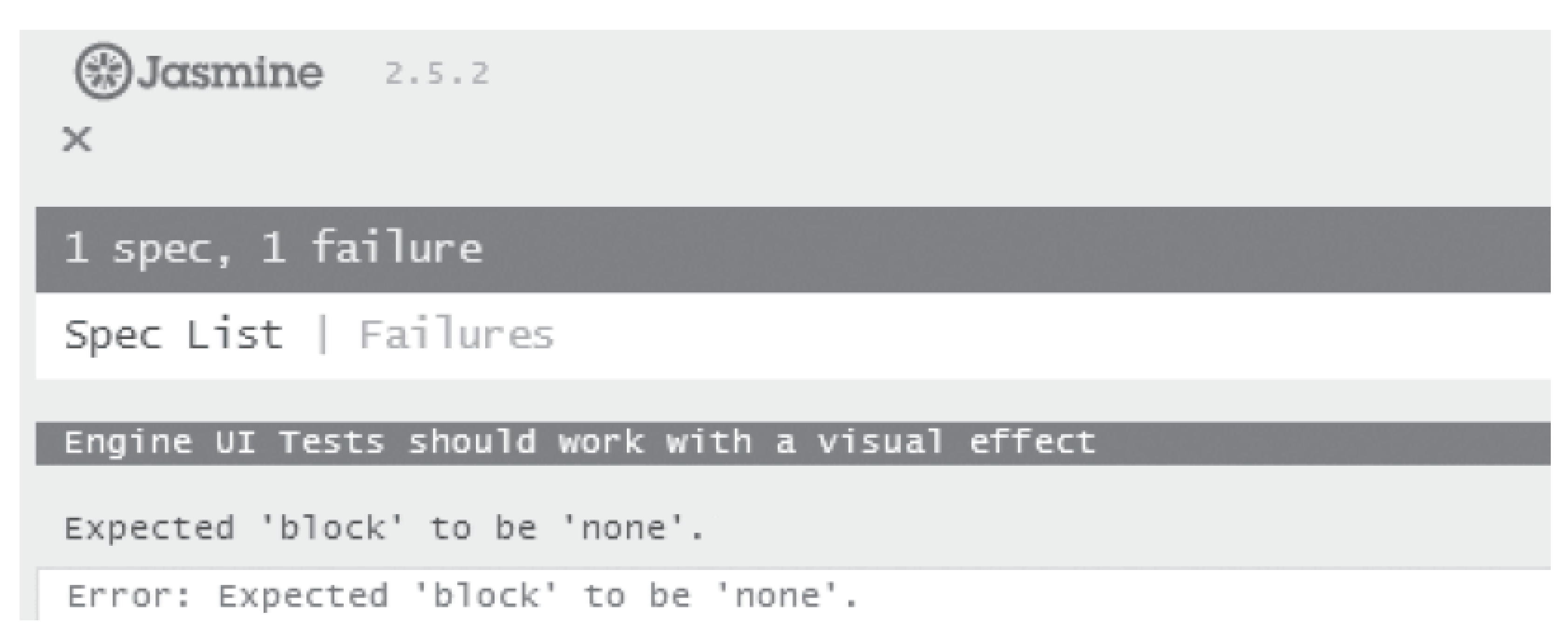


图4-14　Engine单元测试失败报告

测试失败的原因是engine.start里使用的fadeOut是一个异步调用，页面元素要在1秒后才消失。但是在测试用例里，engine.start函数执行后马上检验页面元素是否消失，此时测试页面元素仍旧存在，所以测试失败。例如：

```
        engine.start();
        {…}
        expect(el.css("display")).toBe("none");
```

为了测试以上的异步代码，需要用到Jasmine的异步支持。

4.6.1 Jasmine的异步支持

为了支持异步代码的测试，Jasmine的全局函数beforeAll、afterAll、beforeEach、afterEach和it的回调函数（代码块）有一个可选参数done。例如：

```
describe("Asynchronous specs", function () {
   var value;
   beforeEach(function (done) {
      setTimeout(function () {
         value = 0;
         done();
      }, 1000);
   });
   it("should support async execution", function (done) {
   …
   });
});
```

参数done用于通知Jasmine：这里有一个异步函数，必须等到done()被调用后才能结束当前测试步骤（beforeAll、afterAll、beforeEach、afterEach和it），再继续执行下一步的测试代码。所以在上面的示例中，Jasmine会在beforeEach里等待1秒（1秒后done被调用），然后再执行后续的it函数。

默认情况下Jasmine在每一步骤（beforeAll、afterAll、beforeEach、afterEach和it）里最多等待5秒。如果5秒后done还是没有被调用，当前的测试用例会被标记为失败，然后Jasmine继续执行下一测试步骤。

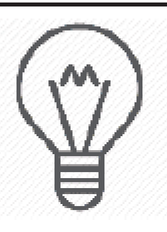
开发人员可以修改全局的默认超时时间（例如修改为2秒），代码如下：

```
jasmine.DEFAULT_TIMEOUT_INTERVAL = 2000;
```

或者在beforeAll、afterAll、beforeEach、afterEach和it这些函数中传入一个额外的超时设置，单独调整这一步骤的超时时间。例如：

```
   beforeEach(function (done) {
```

```
    setTimeout(function () {
        value = 0;
        done();
    }, 1000);
}, 2000);
```



如果想要手工标记测试用例为失败，可以调用done.fail()函数。

了解了Jasmine对测试异步代码的支持后，前面Engine_spec.js里的测试用例就可以改写为：

```
beforeEach(function (done) {
    el = $('#fade-div');
    engine = new Engine(el);
    engine.start(function () {
        done();
    });
});
```

在本示例中，fadeOut（engine.start）可以接受外部输入的回调函数，所以在回调函数里调用done告诉JasminefadeOut已经执行完毕，可以接下来执行下一步骤（it块）进行验证。

如果fadeOut或engine.start不接受外部的回调函数，那么可以使用setTimeout或setInterval延迟一段时间再进行验证①。示例代码如下：

```
it('should work with a visual effect', function (done) {
    setTimeout(function() {
        expect(el.css("display")).toBe("none");
        done();
    }, 2000);
});
```

① sheelc. testing fadeOut() method[OL]. 2014. https://github.com/jasmine/jasmine/issues/516.

4.6.2　模拟JavaScript Timeout相关函数

虽然以上方法可以测试异步代码，但是单元测试需要尽可能地快速运行，如果测试用例中频繁出现setTimeout或setInterval，那么大量时间会被浪费在等待上，造成测试效率低下。为此Jasmine提供了一个虚拟时钟，模拟JavaScript Timeout相关函数。

为了演示Jasmine的虚拟时钟，先在Engine.js里为Engine添加一个新方法：

```
Engine.prototype.pause10seconds = function(cb) {
    setTimeout(cb, 10000);
};
```

虽然可以按前面所述的方案在pause10seconds的回调函数里检验测试结果，但是该方法需要等待10秒，测试效率很低。为了让测试时间“快进”，需要使用Jasmine的虚拟时钟。其用法是：

（1）调用jasmine.clock().install函数安装这个虚拟时钟（通常在beforeEach里）。

（2）调用jasmine.clock().tick函数“快进”时间，这个函数接受的参数是毫秒数。然后验证回调函数的结果。

（3）测试完毕后需要调用jasmine.clock().uninstall函数卸载虚拟时钟（通常在afterEach里）。

测试用例如下：

```
    describe('Clock Tests', function () {
       beforeEach(function () {
          jasmine.clock().install();
       });
       afterEach(function () {
          jasmine.clock().uninstall();
       });
       it('should callback after 10 seconds', function () {
          var engine = new Engine();
          var timerCallback = jasmine.createSpy("timerCallback");
          engine.pause10seconds(timerCallback);
```

```
        jasmine.clock().tick(8000);
        expect(timerCallback).not.toHaveBeenCalled();
        jasmine.clock().tick(2050);
        expect(timerCallback).toHaveBeenCalled();
    });
});
```

以上示例里先使用jasmine.createSpy创建一个spy函数，然后执行被测代码engine.pause10seconds，理论上需要等待10秒spy函数才会被执行。调用jasmine.clock().tick先“快进”8秒，这时验证spy函数没有被调用过。再次调用jasmine.clock().tick“快进”2秒多，此时虚拟时钟已经过了10秒，spy函数执行完毕。

此示例没有使用done，但是利用Jasmine虚拟时钟“快进”了时间，使得setTimeout和setInterval的回调函数以同步的方式被执行。

Jasmine虚拟时钟的“快进”功能极大地提高了单元测试的效率。

4.7 Jasmine插件

Jasmine作为一款流行的JavaScript单元测试框架，拥有大量可以扩展Jasmine功能的插件。本书介绍jasmine-ajax和jasmine-jquery这两个插件。

4.7.1 jasmine-ajax

JavaScript应用经常使用Ajax将数据发送给远程服务器并接收服务器的响应结果。单元测试需要隔离实际的服务器，并且控制Ajax调用的返回结果。为此Jasmine提供插件jasmine-ajax[1]来截获Ajax请求并模拟返回结果。

1. 安装jasmine-ajax库

使用npm命令安装jasmine-ajax库。

```
C:\jasmine-demo>npm install --save-dev jasmine-ajax
```

① Pivotal Labs. jasmine-ajax - A library for faking Ajax responses in your Jasmine suite[OL]. 2015. https://github.com/jasmine/jasmine-ajax.

安装完毕后目录结构如下：

```
-- jasmine-demo
  +-- node_modules
  | +-- jasmine-ajax
  | +-- jasmine-core
  | +-- jquery
  +-- src
  +-- spec
```

2. 引用jasmine-ajax库

jasmine-ajax库文件是mock-ajax.js。为了使用jasmine-ajax插件，需要在测试执行页面jasmine-demo\spec\Plugin\SpecRunner.html里引用mock-ajax.js。其代码如下：

```
<script src="../../node_modules/jasmine-ajax/lib/mock-ajax.js"></script>
```

3. 初始化和卸载jasmine-ajax

如果要测试Ajax调用，需要预先调用jasmine.Ajax.install进行初始化（通常在beforeEach里），替换XMLHttpRequest对象。XMLHttpRequest是现代浏览器内建对象，Ajax调用就是通过XMLHttpRequest对象和后端服务器进行数据交换。XMLHttpRequest对象被替换后，当前页面的Ajax调用都会被jasmine-ajax所截获。

测试完毕后需要调用jasmine.Ajax.uninstall以卸载jasmine-ajax（通常在afterEach中），恢复原有的XMLHttpRequest对象。

```
describe('jasmine-ajax', function () {
   beforeEach(function () {
      jasmine.Ajax.install();
   });
   afterEach(function () {
      jasmine.Ajax.uninstall();
   });
});
```

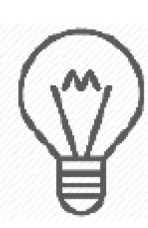
4. 模拟Ajax返回结果

jasmine-ajax截获Ajax请求后，即可用于在任意时间调用respondWith函数返回的结果。示例代码如下：

```
it('should specify response when you need it', function () {
   var doneFn = jasmine.createSpy('success');
   var xhr = new XMLHttpRequest();
   xhr.onreadystatechange = function (args) {
      if (this.readyState == this.DONE) {
         doneFn(this.responseText);
      }
   };
   xhr.open('GET', '/site/test/url');
   xhr.send();
   var request = jasmine.Ajax.requests.mostRecent();
   expect(request.url).toBe('/site/test/url');
   expect(doneFn).not.toHaveBeenCalled();
   request.respondWith({
      'status': 200,
      'contentType': 'text/plain',
      'responseText': 'awesome response'
   });
   expect(doneFn).toHaveBeenCalledWith('awesome response');
});
```

以上示例先创建一个spy函数doneFn作为Ajax调用结束的回调函数，然后测试用例发出一个Ajax请求。当然这个请求会被jasmine-ajax截获，使用jasmine.Ajax.requests.mostRecent获取这个请求对象。示例代码如下：

```
var request = jasmine.Ajax.requests.mostRecent();
```



jasmine.Ajax.requests会返回RequestTracker对象。以上示例使用了mostRecent成员函数，其他成员可以参考mock-ajax.js里的实现完成：

```
function RequestTracker() {
    var requests = [];
    this.track = function(request) {
      requests.push(request);
    };
    this.first = function() {
      return requests[0];
};
...
```

从这个请求对象的属性如url、method、data()等可以获得Ajax请求的相关信息。此时spy函数doneFn还没有被调用，因为此时还没有设置返回结果。

调用请求对象的respondWith函数可以设置返回结果。本示例设置了状态代码（status）、内容类型（contentType）以及返回文本（responseText）。最后验证spy函数doneFn已被调用。

除了利用responseWith函数在需要的时候返回指定内容，也可以预先设置条件。一旦Ajax请求符合这个条件，jasmine-ajax会立即返回预设结果。示例代码如下：

```
it('should allow responses to be setup ahead of time', function () {
   var doneFn = jasmine.createSpy('success');
   jasmine.Ajax.stubRequest('/another/url').andReturn({
      'responseText': 'immediate response'
   });
   var xhr = new XMLHttpRequest();
   xhr.onreadystatechange = function (args) {
      if (this.readyState == this.DONE) {
         doneFn(this.responseText);
```

```
            }
        };
        xhr.open('GET', '/another/url');
        xhr.send();
        expect(doneFn).toHaveBeenCalledWith('immediate response');
    });
```

上面的示例中使用了stubRequest函数设置条件，这样一旦有Ajax请求访问'/another/url'，jasmine-ajax立即返回指定结果，即如下代码的作用。

```
    jasmine.Ajax.stubRequest('/another/url').andReturn({
        'responseText': 'immediate response'
    });
```

4.7.2 jasmine-jquery

除了测试异步代码，前端JavaScript单元测试的另外一个难点是测试DOM操作。其难点表现在以下两个方面：

1）加载并清除测试所需的DOM元素

为了加载测试所需的DOM元素，需要预先在SpecRunner.html文件里准备DOM元素。例如为前面示例测试Engine.start函数准备了需要淡出的元素：

```
<body>
    <div id="fade-div">some content</div>
</body>
```

或者在测试用例里动态添加以下代码：

```
$('body').append('<div id="fade-div">some content</div>');
```

但是这样写代码非常烦琐，而且当测试用例执行完毕后，必须手工清除这些DOM元素，恢复测试环境，以免影响其他测试用例的运行。

2）验证DOM元素的变化

执行被测代码后需要验证DOM元素的变化。示例代码如下：

```
expect(el.css("display")).toBe("none");
```

但是Jasmine内建的断言库并没有提供对DOM的特殊支持。针对这个问题，jasmine-jquery[①]插件提供了API来帮助加载并自动清除HTML、CSS和JSON对象，同时它提供非常强大的自定义匹配器，简化对DOM条件的断言。本书主要介绍HTML对象的加载。如果读者对CSS等技术感兴趣，可以参考此网址https://github.com/velesin/jasmine-jquery的相关内容。

1. 安装jasmine-jquery库

使用npm命令安装jasmine-jquery库。

```
C:\jasmine-demo>npm install --save-dev jasmine-jquery
```

安装完毕后目录结构如下：

```
-- jasmine-demo
  +-- node_modules
  | +-- jasmine-ajax
  | +-- jasmine-core
  | +-- jasmine-jquery
  | +-- jquery
  +-- src
  +-- spec
```

2. 引用jasmine-jquery库

jasmine-jquery的库文件是jasmine-jquery.js。在测试执行页面jasmine-demo\spec\Plugin\SpecRunner.html里引用jasmine-jquery.js的代码如下：

```
<script src="../../node_modules/jasmine-jquery/lib/jasmine-jquery.js"></script>
```

① Wojciech Zawistowski. jQuery matchers and fixture loader for Jasmine framework[OL]. 2015. https://github.com/velesin/jasmine-jquery.

3. 加载HTML DOM元素

在jasmine-demo\src\Plugin\domfunctions.js中准备一个JavaScript函数用来改变DOM元素里的内容，作为被测代码：

```
/* domfunctions.js */
function changeContainerText(txt) {
    if (typeof txt === 'undefined') {
        txt = 'hello world';
    }
    $("#container").text(txt);
}
```

所以在测试执行页面jasmine-demo\spec\Plugin\SpecRunner.html里引用这个js文件：

```
<script src="../../src/Plugin/domfunctions.js"></script>
```

测试这个函数需要准备一个id是container的div。jasmine-jquery提供了多种方式加载DOM元素，如表4-6所示。

表4-6 jasmine-jquery HTML DOM加载函数

全局函数名	描　述
loadFixtures(fixtureUrl[, fixtureUrl,…])	从一个或多个文件中加载DOM元素，并附加到新容器中
appendLoadFixtures(fixtureUrl[, fixtureUrl, …])	和loadFixtures一样，但是会把加载的DOM元素附加在已存在的容器里
readFixtures(fixtureUrl[, fixtureUrl, …])	从一个或多个文件加载DOM元素，但是不把它们加到容器里，而是返回字符串
setFixtures(html)	将HTML代码片段附加到新容器中
appendSetFixtures(html)	将HTML代码片段附加在已存在的容器里

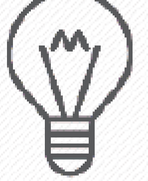


表4-6提到的“容器”指的是执行单元测试时jasmine-jquery会在测试执行页面SpecRunner.html里动态创建id为jasmine-fixtures的div元素：

```
<div id="jasmine-fixtures">    </div>
```

jasmine-jquery加载的DOM元素会被插入到这个div元素里。

这个“容器”在测试用例结束后会被自动清除，不会影响下一个测试用例的运行。

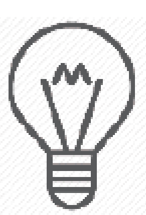


表4-6里的全局函数都是jasmine.getFixtures()返回对象里的成员函数的简化形式：

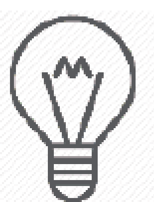
```
loadFixtures() ⟹jasmine.getFixtures().load()
appendLoadFixtures() ⟹jasmine.getFixtures().appendLoad()
readFixtures() ⟹jasmine.getFixtures().read()
setFixtures() ⟹jasmine.getFixtures().set()
appendSetFixtures() ⟹jasmine.getFixtures().appendSet()
```

使用setFixtures函数直接加载HTML代码片段：

```
describe('setFixtures', function () {
  beforeEach(function () {
    setFixtures('<div id="container"></div><button id="btn" onclick="changeContainer
Text()">Click</button>');
  });
});
```

如果HTML片段比较长的话，可以将HTML片段保存到文件中（例如将以上setFixtures函数里的片段保存到jasmine-demo\spec\Fixtures\PluginFixture.html文件），调用loadFixtures函数进行加载：

```
describe('loadFixtures', function () {
  beforeEach(function () {
    var path = '';
    if (typeof window.__karma__ !== 'undefined') {
      path += 'base';
    }
    jasmine.getFixtures().fixturesPath = path + '/spec/Fixtures';
    loadFixtures('PluginFixture.html');
  });
});
```



jasmine-jquery默认会从spec/javascripts/fixtures目录加载HTML文件。这个默认路径可以被修改，例如：

```
jasmine.getFixtures().fixturesPath = 'my/new/path';
```

注意：如果在单元测试里使用Karma的话，测试文件会从base/目录里被访问，所以修改jasmine-jquery默认路径时需要加上base/前缀：

```
jasmine.getFixtures().fixturesPath = 'base/my/new/path';
```

loadFixture函数会使用Ajax调用来加载HTML文件，因为浏览器默认不允许Ajax加载本地文件。如果在本地直接打开SpecRunner.html文件，以上的示例将无法运行，所以需要使用一个HTTP服务器，并且将HTTP服务器的根目录设为jasmine-demo。这样通过HTTP服务器访问SpecRunner.html，即可使loadFixture成功加载/Spec/Fixtures/PluginFixture.html。

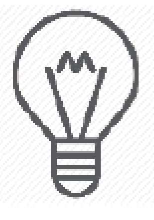
使用npm命令可以全局安装一个简易的HTTP 服务器：

```
npm install http-server -g
```

然后在命令控制台将当前目录切换到jasmine-demo，运行以下命令：

```
http-server
```

这个简易HTTP服务器默认使用8080端口，所以需访问http://localhost:8080/spec/plugin/specrunner.html进行单元测试。



jasmine-ajax会截获所有的Ajax请求，而jasmine-jquery的loadFixture会调用Ajax加载HTML文件，所以当两者一起使用的时候，必须先调用loadFixture，然后再初始化jasmine-ajax。例如：

```
beforeEach(function(){
  // first load your fixtures
  loadFixtures('fixture.html');
  // then install the mock
  jasmine.Ajax.install();
});
```

4. 验证DOM元素的变化

jasmine-jquery为jQuery框架提供了大量自定义匹配器，如表4-7所示。为了测试以上示例里changeContainerText函数是否更改DOM成功，可以使用以下的测试用例：

```
it('container should have hello world text', function () {
    expect($('#btn')).toExist();
    $('#btn').trigger('click');
    expect($('#container')).toHaveText('hello world');
});
```

表4-7 常用jasmine-jquery自定义匹配器

匹 配 器	描 述
toBeChecked()	验证元素是否被选中，只针对有checked属性的元素。例如： expect($('<input type="checkbox" checked="checked"/>')).toBeChecked()
toBeDisabled()	验证元素是否被禁用
toBeEmpty()	验证元素是否有子元素或内容
toBeHidden()	验证元素是否被隐藏。以下几种情况可以认为元素被隐藏： • CSS的display值为none • form元素的类型为hidden • 元素的宽度和高度都设为0 • 上级元素是隐藏的，所以元素不会被显示
toBeVisible()	验证元素是否可见
toContain(string)	验证元素是否包含指定字符串。例如： expect($(' ')) .toContain('some-class')
toExist()	验证元素是否存在
toHandle(eventName)	验证元素是否能处理指定事件。例如： e.g. expect($form).toHandle("submit")
toHaveClass(className)	验证元素是否有指定的CSS class。例如： expect($('<div class="some-class"></div>')).toHaveClass("some-class")
toHaveText(string)	验证元素是否有指定文本或符合正则表达式

读者可以访问https://github.com/velesin/jasmine-jquery了解更多的jasmine-jquery匹配器。

4.8 基于浏览器调试

使用Jasmine进行单元测试时，测试用例是运行在浏览器里的，所以测试用例代码和前端JavaScript代码一样，都可以使用浏览器进行调试。

现代浏览器都内建开发者工具，帮助开发者调试前端程序。以Chrome为例，可以使用快捷键Shift+Ctrl+I启动开发者工具。（启动IE开发者工具的快捷键是F12）

Chrome开发者工具除了具有断点设置、删除，单步调试等基本功能以外，还在Source标签页（参见图4-15）提供了以下几种常用的调试功能：

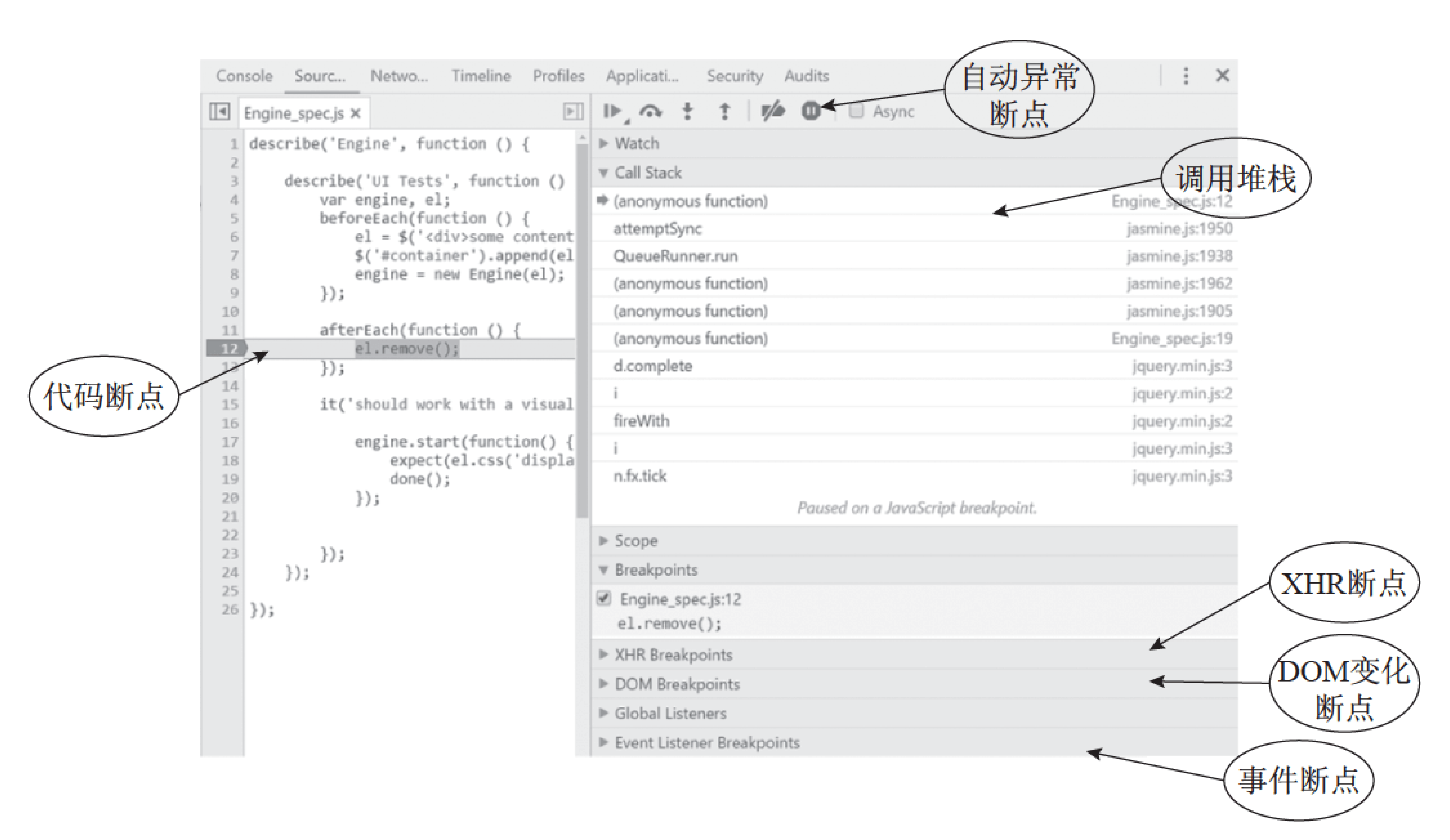


图4-15　Chrome开发者工具Source标签页

1）自动异常断点

这个功能开启后，当JavaScript代码发生异常时，会在异常发生处暂停运行，供开发人员查找异常产生的原因。

2）DOM变化和XHR断点

这两项功能可以对DOM结构改变/属性改变等设置断点，也可以对Ajax请求设置断点。

3）事件断点

此功能可对某个事件（例如鼠标单击）设置断点。

4）调用堆栈分析

此功能可列出当前的调用堆栈。

5）实时代码编辑

代码编辑区可用于在运行时动态改变 JavaScript代码。

在进行JavaScript应用开发的过程中，开发人员会添加必要的调试日志输出语句，方便进行问题定位。这些日志信息可以在Console 标签页内找到，并且能通过单击该日志的末端文件链接直接跳转到Source标签页的源文件中，极大方便了相关代码的定位。

Network标签页（参见图4-16）列出了所有的HTTP请求，以方便用户查看请求内容、HTTP头、请求时间等信息。

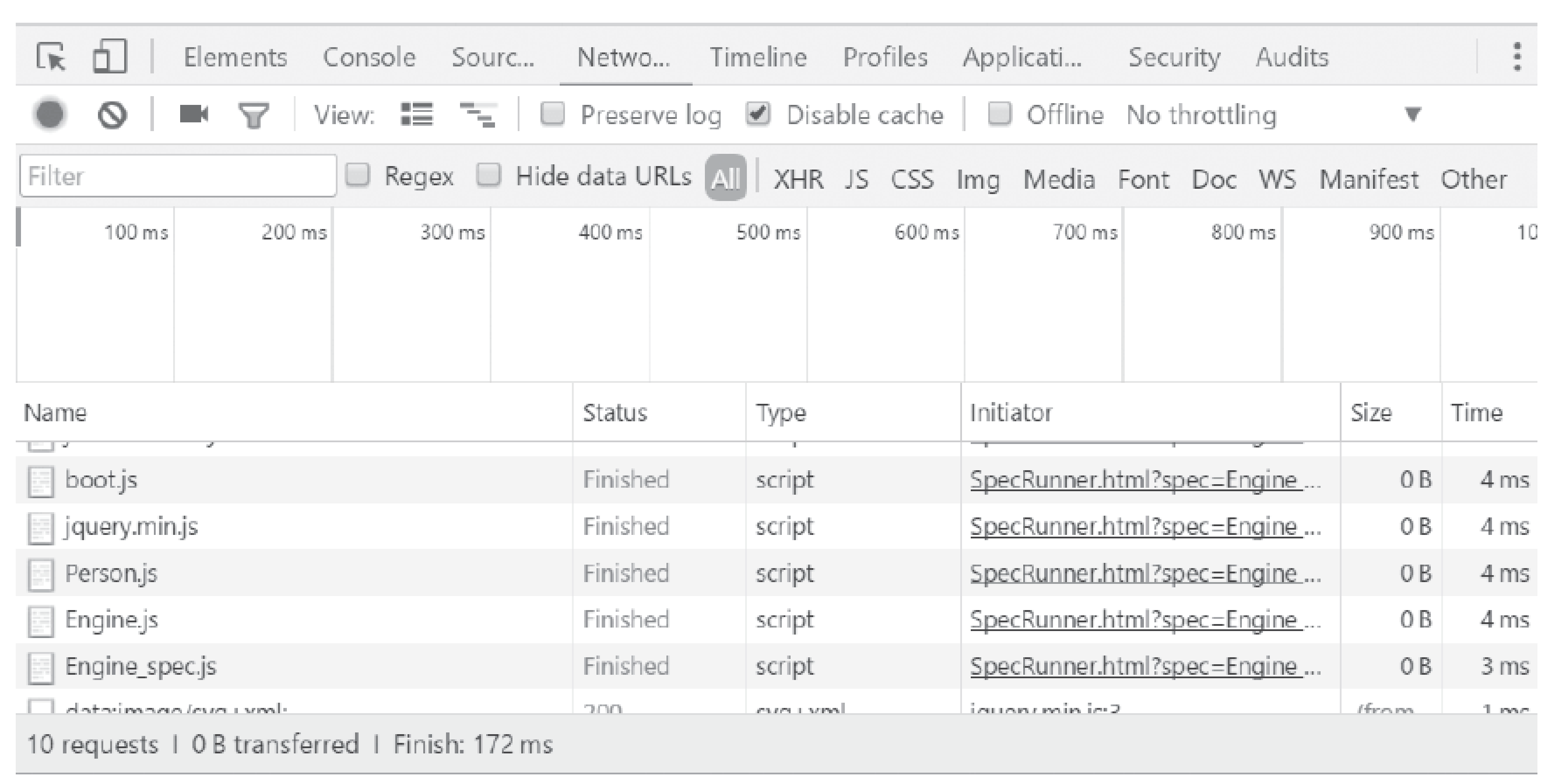


图4-16　Chrome开发者工具Network标签页

如果想要了解更多Chrome开发者工具的信息，可以访问https://developers.google.com/web/tools/chrome-devtools/。

第5章 单元测试执行工具Karma

前端开发人员进行JavaScript单元测试，一般需要执行以下步骤：添加或修改测试用例代码；维护测试执行页面（例如Jasmine里的SpecRunner.html），当添加了一个新的JavaScript源文件时，需要在SpecRunner.html里引用该文件和相应的测试文件；手动刷新浏览器（有时候还需要清空浏览器缓存），如果JavaScript代码要在不同浏览器内测试，那么需要分别刷新各个浏览器；在浏览器内查看测试结果。

这个过程需循环反复，使得开发人员在无关代码的工作上花费太多精力，降低了工作效率。测试执行工具可帮助开发人员从维护SpecRunner.html、刷新浏览器等机械重复的事情中解放出来，真正把关注点放到编写测试、运行测试、重构等工作上来。

本章将介绍流行的测试执行工具Karma①，内容包括：

- 初识Karma
- 安装Karma和相关插件
- Karma的配置
- 基于Karma的调试
- 前端自动化任务构建工具
- Karma和gulp集成

5.1 初识Karma

Karma是Google为AngularJS开发的测试执行工具。它不仅可以测试AngularJS程序，还

① Friedel Ziegelmayer. Karma - Spectacular Test Runner for Javascript[OL]. [2016]. http://karma-runner.github.io.

被广泛用于测试前端JavaScript程序。

Karma的设计目的主要有以下4点：

- 快速高效
- 可靠
- 运行于真实的浏览器里
- 无缝的使用流程

Karma是一个典型的C/S架构程序，包括服务器和客户端，如图5-1所示。

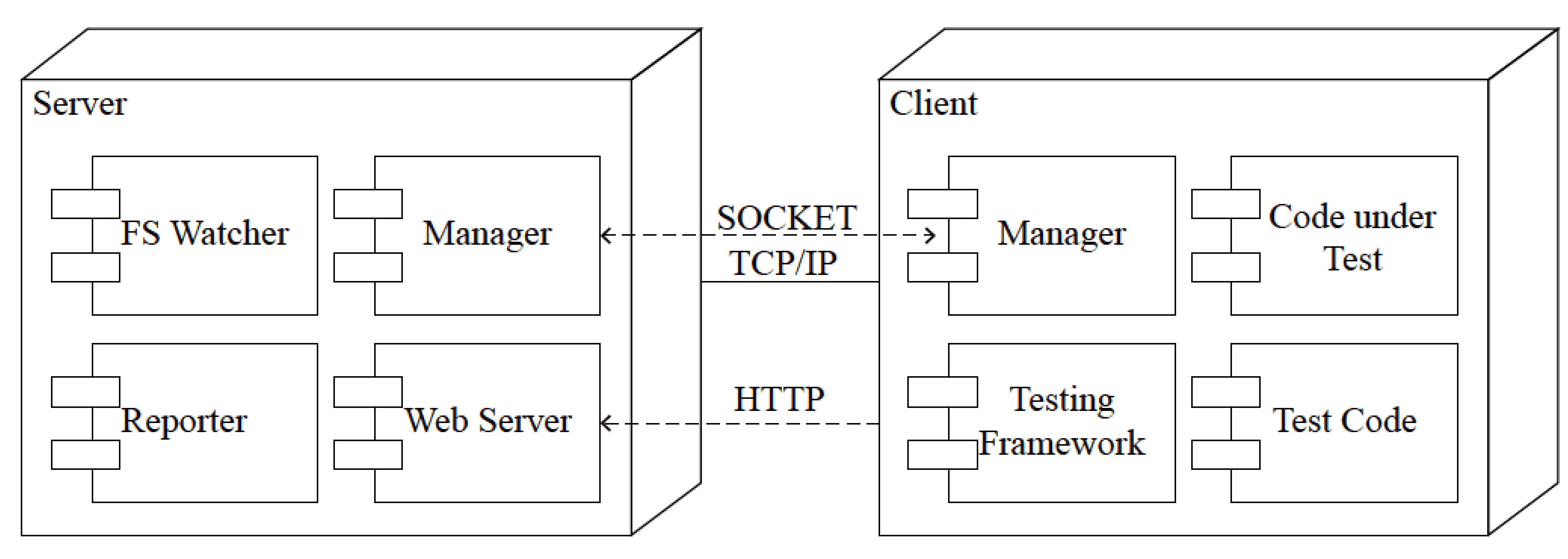


图5-1　Karma架构

运行的时候，Karma会启动一个基于Node.js的Web服务器程序。浏览器（客户端）可以访问Karma服务器监听的URL（默认是http://localhost:9876/）和服务器建立连接。这些和服务器建立了连接的浏览器就成为了受控的客户端。

根据测试配置文件，Karma服务器能向客户端提供所需的测试文件，让测试用例在所有受控的客户端运行。测试结束后，客户端将测试用例执行结果传回服务器，由服务器输出结果给开发人员（默认在命令控制台）。使用Karma，开发人员不需要维护SpecRunner.html，也不用离开集成开发环境去各个浏览器查看测试结果。

服务器会监视本地源码或测试文件的更改。一旦有更新，它会通知受控的客户端（浏览器），让客户端重新加载测试代码进行测试。这样开发人员就无需再手动来刷新浏览器了。

简而言之，使用Karma，开发人员可以简单而又快速地在不同浏览器中进行自动化单元测试。如果想要了解更多Karma的设计思想，请参阅Karma作者的硕士论文[①]。

---

① Vojtech Jína. JavaScript Test Runner[OL]. 2013. https://github.com/karma-runner/karma/raw/master/thesis.pdf.

5.2 安装Karma和相关插件

5.2.1 安装Karma

Karma是Node.js程序，所以需在项目目录下使用npm命令进行本地安装：

```
C:\jasmine-demo>npm install --save-dev karma
```

安装完毕后可以在命令控制台输入以下命令启动Karma：

```
C:\jasmine-demo>node .\node_modules\karma\bin\karma start
26 11 2016 14:52:54.758:WARN [karma]: No captured browser, open http://localhost:9876/
26 11 2016 14:52:54.769:INFO [karma]: Karma v1.3.0 server started at http://
localhost:9876/
```

使用浏览器访问http://localhost:9876，和服务器建立连接，如图5-2所示，然后就可以接受并执行服务器向它发送的指令（在命令控制台按快捷键Ctrl+C可停止服务器）。

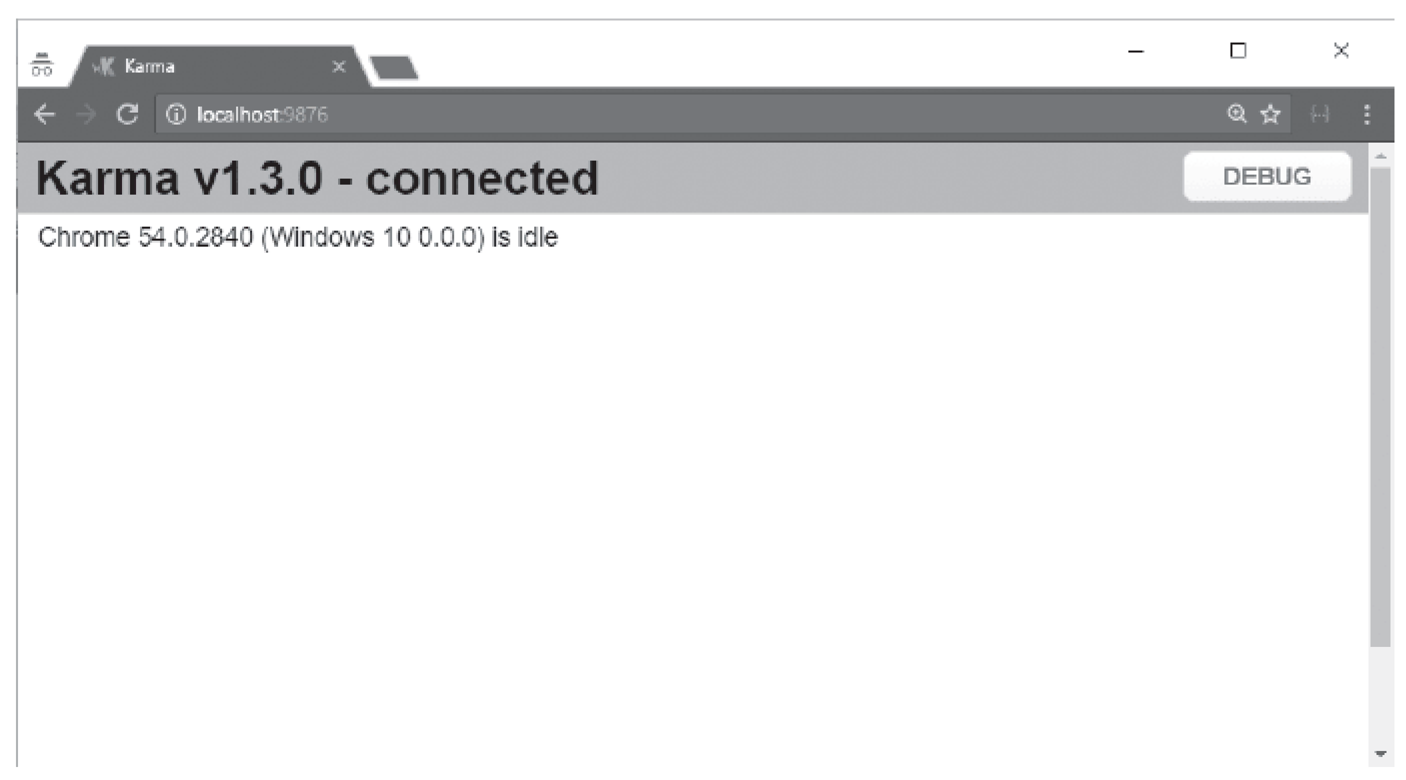


图5-2　受控的Karma客户端（浏览器）

由于调用node .\node_modules\karma\bin\karma来执行Karma不太方便，所以通常会使用下面的命令安装一个全局的命令行接口：

```
C:\jasmine-demo>npm install karma-cli -g
```

安装了全局的命令行接口后可以直接执行以下命令启动Karma服务器：

```
C:\jasmine-demo>karma start
```

5.2.2　安装插件

Karma的扩展性好，很多功能是通过插件方式实现的。插件是npm软件包，所以要使用npm命令安装插件：

```
npm install karma-<plugin name> --save-dev
```

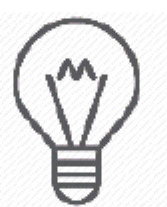
Karma启动时会加载所有名字以karma-开头的npm软件包。

1. 浏览器启动器

Karma在启动服务器时，可以同时通过浏览器启动器插件启动浏览器[①]，这些浏览器和服务器建立连接，加载并运行测试用例，它们也是要测试的目标环境。本书示例安装的是Chrome和Firefox的启动器，命令如下：

```
C:\jasmine-demo> npm install karma-chrome-launcher --save-dev

C:\jasmine-demo> npm install karma-firefox-launcher --save-dev
```



访问网址https://www.npmjs.com/browse/keyword/karma-launcher可以了解更多的启动器插件。

浏览器启动器插件使用默认路径寻找浏览器的执行文件。如果浏览器未安装在默认路径处，需要手动设置环境变量<BROWSER>_BIN，例如：

```
C:> SET FIREFOX_BIN=C:\Program Files (x86)\Mozilla Firefox\firefox.exe
```

2. 测试框架适配器

Karma支持Jasmine、QUnit、Mocha等多种测试框架。为了使用这些测试框架，除了安装测试框架本身的类库外，还需要安装相应测试框架的适配器。本书示例使用Jasmine，所以安装Jasmine适配器，命令如下：

```
C:\jasmine-demo> npm install karma-jasmine --save-dev
```

3. 测试报告插件

如果想让Karma输出指定格式的测试结果，例如JUnit格式和HTML格式，需要安装相

① Friedel Ziegelmayer. Karma - Browsers[OL]. [2016]. http://karma-runner.github.io/1.0/config/browsers.html.

应的插件，命令如下：

```
C:\jasmine-demo> npm install karma-junit-reporter --save-dev
C:\jasmine-demo> npm install karma-html-reporter --save-dev
```

JUnit是一种基于XML的报告格式，被广泛应用于各种持续集成系统内，但是XML格式不适合直接阅读，所以一般同时输出HTML格式的报告。

5.3 Karma的配置

5.3.1 生成配置文件

虽然安装了Karma，但它还不知道该做些什么，开始测试之前，开发人员需要在项目目录下创建一个配置文件对Karma进行相关设置，命令如下：

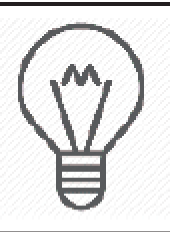
```
C:\jasmine-demo> karma init
```

和npm init类似，这个命令采用互动方式，要求用户回答一些问题。本示例使用的都是默认设置：

```
Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next question.
> jasmine
Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options. Enter to move to the next question.
> no
Do you want to capture any browsers automatically ?
Press tab to list possible options. Enter empty string to move to the next question.
> Chrome
>
```

```
What is the location of your source and test files ?
You can use glob patterns, eg. "js/*.js" or "test/**/*Spec.js".
Enter empty string to move to the next question.
>
Should any of the files included by the previous patterns be excluded?
You can use glob patterns, eg. "**/*.swp".
Enter empty string to move to the next question.
Do you want Karma to watch all the files and run the tests on change ?
Press tab to list possible options.
> yes
Config file generated at "C:\jasmine-demo\karma.conf.js".
```

问题回答完毕后会在当前目录下生成karma.config.js文件。



karma.config.js是默认的配置文件。如果想要让karma init生成其他配置文件，例如my.config.js，可以执行命令karma init my.conf.js。

配置文件默认使用的是JavaScript代码，也可以用CoffeeScript代码编写。如果运行karma init，后面跟一个*.coffee扩展的文件名，例如karma init karma.conf.coffee，就会生成一个CoffeeScript文件。

5.3.2　配置文件的说明

首先运行karma start命令启动Karma服务器，这同时会启动一个Chrome浏览器和服务器连接，开始进行单元测试。执行命令及输出的结果如下：

```
C:\jasmine-demo>karma start
26 11 2016 20:19:04.768:WARN [karma]: No captured browser, open http://localhost:9876/
26 11 2016 20:19:04.778:INFO [karma]: Karma v1.3.0 server started at http://
localhost:9876/
26 11 2016 20:19:04.779:INFO [launcher]: Launching browser Chrome with unlimited
concurrency
```

```
26 11 2016 20:19:04.789:INFO [launcher]: Starting browser Chrome
26 11 2016 20:19:06.671:INFO [Chrome 54.0.2840 (Windows 10 0.0.0)]: Connected on socket
/#K3zK1lyRBX4BPSPHAAAA with id 65015763
Chrome 54.0.2840 (Windows 10 0.0.0): Executed 0 of 0 ERROR (0.002 secs / 0 secs)
```

以上的输出结果里显示没有测试用例被执行，这是因为还没有告诉Karma需要测试哪些代码。现在对配置文件karma.config.js做一些修改。



默认情况下，karma会依次寻找下列配置文件：

- ./karma.conf.js
- ./karma.conf.coffee
- ./.config/karma.conf.js
- ./.config/karma.conf.coffee

如果需要让karma start使用其他配置文件，例如my.config.js，可以执行karma start my.conf.js命令。

1. 常用配置项

karma.config.js本身是一个Node.js模块，内容如下：

```
module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine'],
…
    })
}
```

该模块调用config.set函数对Karma进行设置。config.set函数输入参数是一个对象，其各个字段就是用户可以调整的Karma的配置。常用的配置项如表5-1所示。

表5-1　常用Karma配置项

配　置　项	类型	默　认　值	描　　述
basePath	字符串	“”	根目录，用来解析定义在files和exclude里的相对路径。如果basePath本身是一个相对路径，那么它会被解析为相对于配置文件的 __dirname
frameworks	数组	[]	要使用的测试框架列表。通常，用户会设定为['jasmine']、['mocha']或['qunit']。注意：这里设置的所有框架都需要额外安装插件/框架库

配　置　项	类型	默　认　值	描　　述
files	数组	[]	需要被加载到浏览器的文件列表和文件模式列表，告诉Karma需要测试哪些文件。通常指单元测试所需要的源文件，测试用例和它们的依赖文件 例如 ['src/**/*.js', 'spec/**/*.js']，表示需要加载src和spec目录及子目录下的所有js文件
exclude	数组	[]	需要从files中排除掉的文件列表
preprocessors	对象	{'**/*.coffee': 'coffee'}	需要做预处理的文件，以及这些文件对应的预处理器。此处设置如何转换CoffeeScript、ES6 等代码。注意：这些预处理器（除了CoffeeScript）都需要额外安装类库
reporters	数组	['progress']	测试结果报告插件列表。本书实例使用的是['progress', 'junit', 'html']
port	数字	9876	Web服务器所监听的端口
colors	布尔值	true	在输出内容（报告插件和日志）中启用/禁用颜色
logLevel	常量	config.LOG_INFO	日志记录级别。可用值为： config.LOG_DISABLE config.LOG_ERROR config.LOG_WARN config.LOG_INFO config.LOG_DEBUG
autoWatch	布尔值	true	开启或关闭监视文件功能。功能开启时，当文件发生变化时自动执行测试
browsers	数组	[]	Karma自动启动的浏览器列表，也就是要测试的目标环境。当Karma启动时，它会同时启动列表里的浏览器；当Karma关闭时，它也会关闭列表里的浏览器。如果不设置这个配置项，也可以手动启动浏览器：访问Karma监听的URL，使得该浏览器成为Karma的受控客户端，接收Karma的指令 注意：要想让Karma启动列表里的浏览器，需要额外安装相应的浏览器启动器 本书实例使用的是['Chrome', 'Firefox']，更多内容请参阅以下网址： http://karma-runner.github.io/1.0/config/browsers.html
singleRun	布尔值	false	持续集成模式。如果设为true，则Karma会启动并控制浏览器，运行测试，然后退出
concurrency	数字	Infinity	设置Karma可以同时启动浏览器的数量

如果要了解更多的配置项，可以访问网址http://karma-runner.github.io/1.0/config/configuration-file.html。

2. Karma加载的文件

Karma不使用SpecRunner.html文件，所以需要人为告诉Karma哪些文件要被加载到浏览器里进行测试。这个工作由配置项files数组完成。

files配置项是文件模式（file pattern）列表。如果文件模式是相对路径，Karma会根据basePath进行解析。如果basePath本身是相对路径，那么它会根据配置文件所在的目录进行

解析。最终，所有的模式都会通过glob[①]对应到文件。开发人员可以使用minimatch[②]表达式作为文件模式来匹配和定位文件。例如：

- **/*.js，所有子目录中以js结尾的文件。
- **/!(jquery).js，同上，但是不包括“jquery.js”文件。
- **/(foo|bar).js，所有子目录中的“foo.js”或“bar.js”文件。

数组里各项模式的顺序决定了文件在浏览器里被加载的顺序。如果多个文件匹配到同一个模式，则按字母顺序加载。每个文件只能被加载一次，如果同一文件被多个模式匹配到，那么文件由第一个匹配到的文件模式加载。

文件模式可以是一个字符串，也可以是一个有以下5个属性的对象：

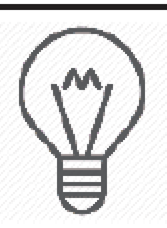
- pattern：需要匹配的文件模式字符串，必须提供这个属性。
- watched：布尔值，默认值是true。如果autoWatch值为true，所有watched设为true的文件都会被Karma监视变化。
- included：布尔值，默认值是true。文件是否应该被浏览器用<script>标签引用加载。如果文件是程序以手动方式加载的，例如通过require.js，则可将该值设为false。
- served：布尔值，默认值是true。文件是否可以通过Karma的Web服务器访问。
- nocache：布尔值，默认值是false。每次请求Karma的Web服务器是否都要到磁盘上读取。

本书示例中使用的是以下files配置项：

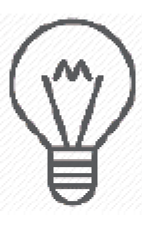
```
files: [
  {pattern: 'spec/Fixtures/*.html', included: false, served: true, watched: false,
nocache: true},
  'node_modules/jquery/dist/jquery.js',
  'node_modules/jasmine-jquery/lib/jasmine-jquery.js',
  'node_modules/jasmine-ajax/lib/mock-ajax.js',
  'src/**/*.js',
  'spec/**/*.js'
],
```

① Isaac Z. Schlueter. glob functionality for node.js[OL]. [2016]. https://github.com/isaacs/node-glob.
② Isaac Z. Schlueter. a glob matcher in javascript[OL]. [2016]. https://github.com/isaacs/minimatch.

其中，spec/Fixtures/*.html是使用jasmine-jquery的loadFixtures函数加载的测试所需的所有HTML文件。用户可以通过http://localhost:9876/base/spec/Fixtures/[MY HTML].html获取该文件。注意URL里的base，karma的Web服务器将服务内容放在了/base 虚拟目录下。



目前版本的Karma Web服务器即使是在Windows系统下也是大小写敏感的，所以一定要注意文件名称的大小写。



本书示例里直接将jquery.js、jasmine-jquery.js和mock-ajax.js添加到files配置项，让Karma直接加载。还有一种方法是使用相应的适配器，例如：

- karma-jquery：https://www.npmjs.com/package/karma-jquery。
- karma-jasmine-ajax：https://www.npmjs.com/package/karma-jasmine-ajax。
- karma-jasmine-jquery：https://www.npmjs.com/package/karma-jasmine-jquery。

虽然适配器本身包含相应的类库（例如karma-jquery包含jQuery库），但是这些适配器里包含的类库版本和测试代码使用的类库版本可能不一致，所以本书示例直接引用这些类库。

3. 测试报告设置

本书示例安装了两个测试报告插件karma-junit-reporter和karma-html-reporter，用于生成JUnit和HTML格式的结果报告。这两个报告插件需要被添加在reporters配置项里，代码如下：

```
reporters: ['progress', 'junit', 'html'],
```

它们还有自己的配置项，例如：

```
junitReporter: {
      outputDir: './report_output',
      outputFile: 'junitreport.xml',
      useBrowserName: false
},
htmlReporter:{
```

```
    outputDir: './report_output',
    reportName: 'unit',
    namedFiles: true
  },
```

这些配置项的解释如下：

- outputDir：报告目录。
- outputFile：JUnit报告文件名。
- useBrowserName：默认情况下JUnit报告保存为$outputDir/$browserName/$outputFile，当useBrowserName值设为false时，不生成$browserName子目录。
- reportName：HTML报告名。
- namedFiles：如果其值设为true，reportName就是报告文件名；如果设为false，则reportName是子目录，最后生成的报告为./report_output/unit/index.html。

要了解更多关于测试报告插件的设置，请参阅以下网址：

https://github.com/karma-runner/karma-junit-reporter

https://github.com/dtabuenc/karma-html-reporter

4. karma.conf.js完整示例

本书示例的karma.conf.js的完整代码如下：

```
// Karma configuration
// Generated on Sat Nov 26 2016 17:38:03 GMT+0800 (China Standard Time)
module.exports = function (config) {
  config.set({
    // base path that will be used to resolve all patterns
    // (eg. files, exclude)
    basePath: '',
    // frameworks to use
    // available frameworks:
    // https://npmjs.org/browse/keyword/karma-adapter
    frameworks: ['jasmine'],
    // list of files / patterns to load in the browser
```

```
    files: [
      {pattern: 'spec/Fixtures/*.html', included: false, served: true, watched: false,
nocache: true},
      'node_modules/jquery/dist/jquery.js',
      'node_modules/jasmine-jquery/lib/jasmine-jquery.js',
      'node_modules/jasmine-ajax/lib/mock-ajax.js',
      'src/**/*.js',
      'spec/**/*.js'
    ],
    // list of files to exclude
    exclude: [
    ],
    // preprocess matching files before serving them to the browser
    // available preprocessors:
    // https://npmjs.org/browse/keyword/karma-preprocessor
    preprocessors: {
    },
    // test results reporter to use
    // possible values: 'dots', 'progress'
    // available reporters:
    // https://npmjs.org/browse/keyword/karma-reporter
    reporters: ['progress', 'junit', 'html'],
    junitReporter: {
          outputDir: './report_output',
          outputFile: 'junitreport.xml',
          useBrowserName: false
    },
    htmlReporter:{
       outputDir: './report_output',
```

```
      reportName: 'unit',
      namedFiles: true
    },
    // web server port
    port: 9876,
    // enable / disable colors in the output (reporters and logs)
    colors: true,
    // level of logging
    // possible values: config.LOG_DISABLE || config.LOG_ERROR
    // || config.LOG_WARN || config.LOG_INFO || config.LOG_DEBUG
    logLevel: config.LOG_INFO,
    // enable / disable watching file and executing tests whenever any file changes
    autoWatch: true,
    // start these browsers
    // available browser launchers:
    // https://npmjs.org/browse/keyword/karma-launcher
    browsers: ['Chrome', 'Firefox'],
    // Continuous Integration mode
    // if true, Karma captures browsers, runs the tests and exits
    singleRun: false,
    // Concurrency level
    // how many browser should be started simultaneous
    concurrency: Infinity
  })
}
```

在项目根目录中执行karma start命令，结果将显示在命令控制台：总共有46个测试用例，略过4个被禁用的测试用例，在Chrome和Firefox中分别执行42个，如图5-3所示。

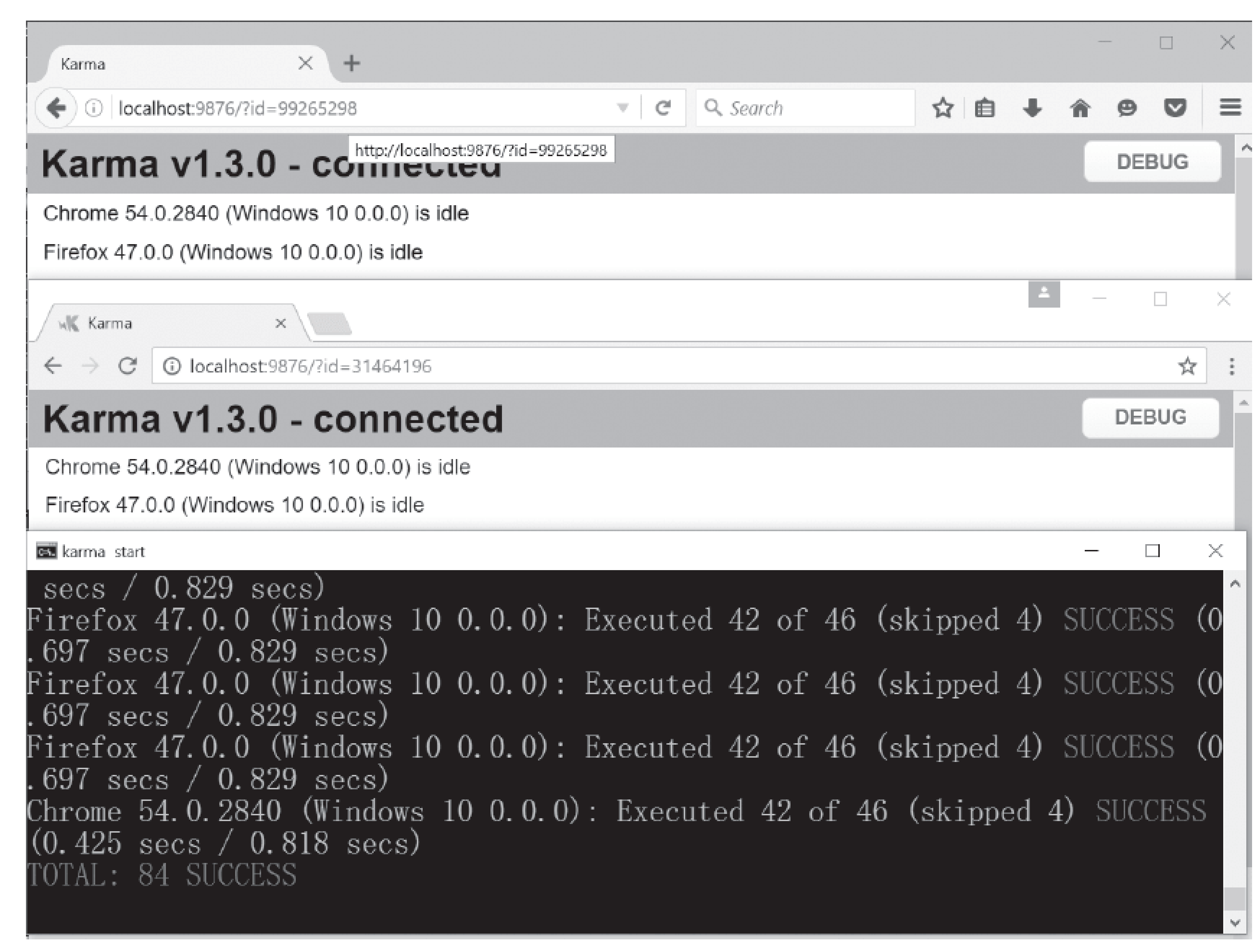


图5-3　Karma命令控制台输出的结果

Karma运行结束后会在C:\jasmine-demo\report_output目录下生成相关格式的报告。

5.4　基于Karma的调试

在Karma环境下调试前端JavaScript代码与测试用例代码和通常的前端调试略有不同。在Karma环境下调试代码的步骤如下：

（1）确保配置文件里的singleRun配置项为false，这样运行测试后浏览器不会自动关闭。

（2）运行karma start命令启动Karma服务器以及浏览器。

（3）单击浏览器页面里的DEBUG按钮，切换至调试模式。（或者直接访问网址http://localhost:9876/debug.html）

（4）启动浏览器里的开发者工具，在代码里设置断点。

（5）刷新页面，触发断点。

（6）接下来就可以进行单步执行、查看变量等常规的调试工作，如图5-4所示。

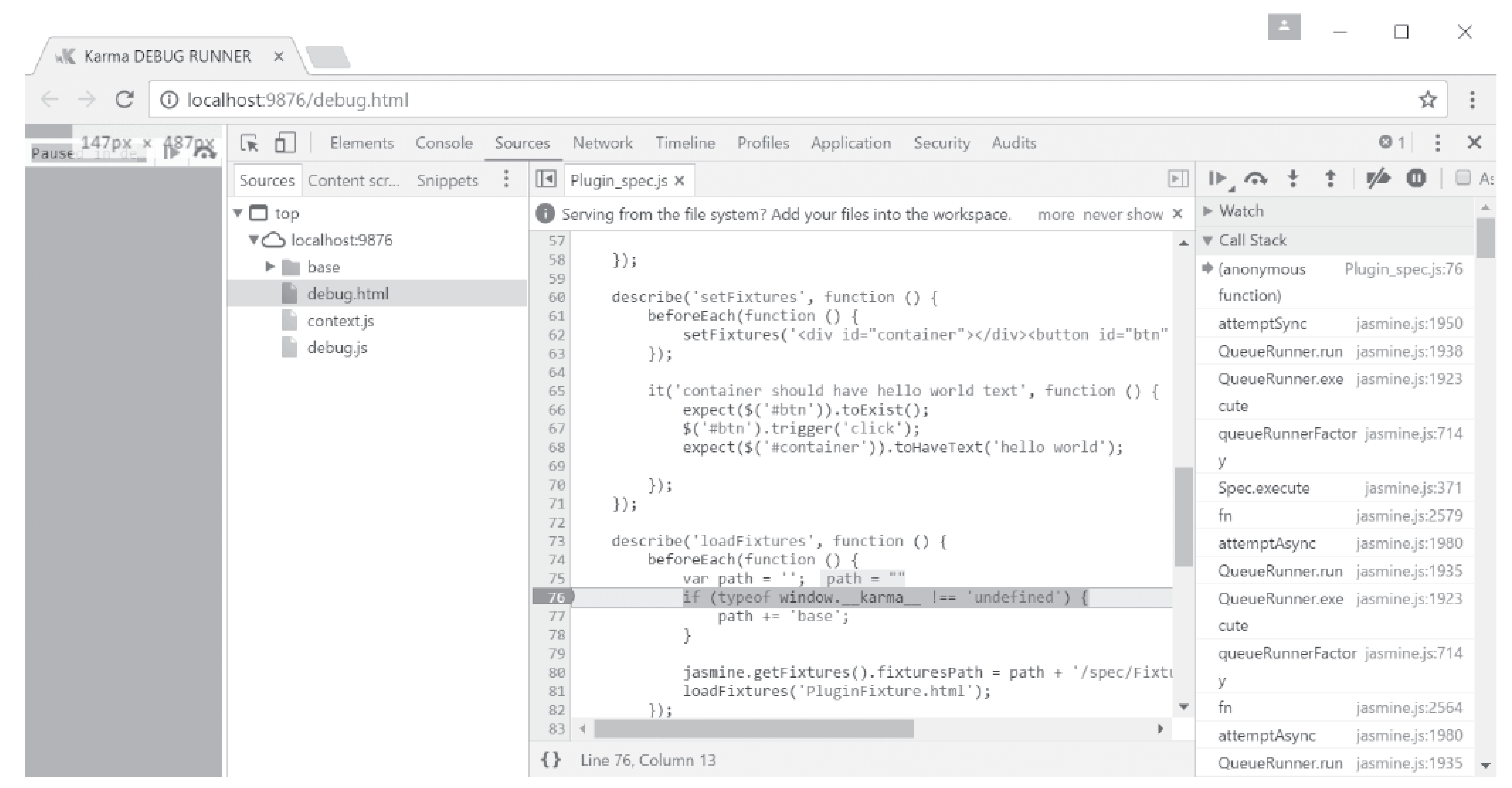


图5-4　基于Karma的调试

5.5　前端自动化任务构建工具

使用Karma进行单元测试只是软件自动化构建的其中一个任务，整个自动化构建流水线如图5-5所示。

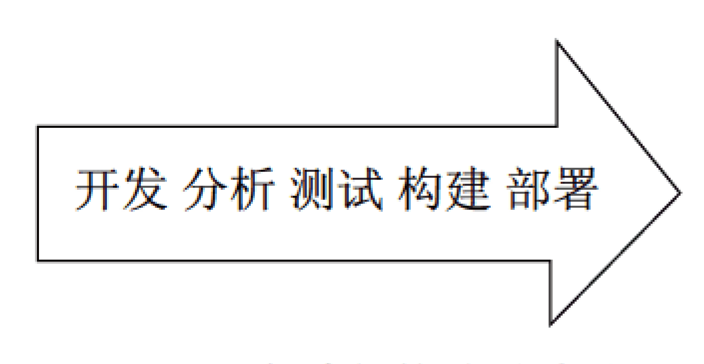


图5-5　自动化构建流水线

前端开发人员平时需要重复进行测试、检查、合并、压缩、格式化、浏览器自动刷新、部署文件生成等步骤。所以借助前端开发自动化任务构建工具，开发人员只要在配置文件里正确设置好任务，任务构建工具就能自动完成大部分重复的常见任务，从而简化工作，提升开发效率。

5.5.1　gulp和Grunt

如今提到前端自动化任务构建工具，总会想起gulp和Grunt。这两个工具都是基于

Node.js的，主要特点如表5-2所示。

表5-2　Grunt和gulp比较

Grunt	gulp
一切任务基于配置（配置即任务），配置比较复杂	代码优于配置。任务通过代码设置，可读性较高
I/O操作基于临时文件	I/O操作基于流（stream based）
约6000个插件（本书写作时）	2700多个插件（本书写作时）

本书使用gulp作为前端自动化任务构建工具。gulp相较于Grunt主要有以下优点：

1. 易用

gulp比Grunt更简洁，而且遵循代码优于配置策略，维护gulp更像是写代码。例如：

```
gulp.task('js', function () {
   return gulp
      .src('**/*.js')
      .pipe(jshint())
      .pipe(concat('all.js'))
      .pipe(uglify())
      .pipe(gulp.dest('./build/'));
});
```

gulp借鉴了UNIX操作系统的管道（pipe）思想，前一级的输出，直接变成后一级的输入，使得它在操作上非常简单。

2. 高效

使用Grunt的I/O过程中会产生一些中间态的临时文件，一些任务生成临时文件，其他任务可能会基于临时文件再做处理并生成最终的构建后文件。频繁在磁盘中读写临时文件，使得构建性能较低，如图5-6所示。

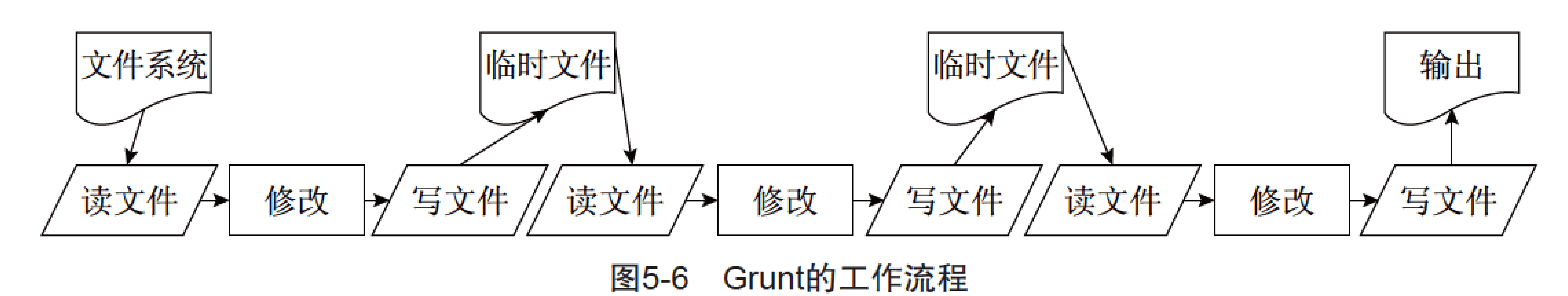


图5-6　Grunt的工作流程

而gulp基于流的方式进行文件处理，通过管道将多个任务和操作连接起来，不需要写中间文件，如图5-7所示。

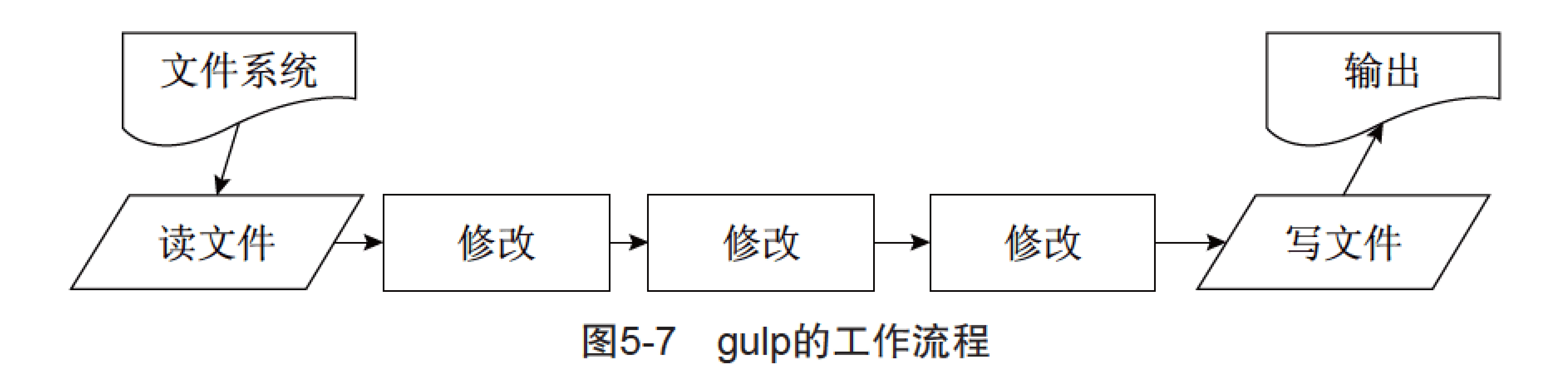


图5-7　gulp的工作流程

3. 高质量

gulp的每个插件只完成一个功能，确保了插件的简单，各个功能通过流方式进行整合并完成复杂的任务。

4. 易学

gulp的API只有4个，掌握了这4个API之后，便可以通过管道流组合出自己想要完成的任务。



“gulp”还是“Gulp”？

gulp一直都是小写的，只在gulp的商标中用大写[①]。

5.5.2　gulp的API

gulp只有4个API，即gulp.task、gulp.src、gulp.dest和gulp.watch。在gulp项目的配置文件gulpfile.js里将使用这些API。

1. gulp.task(name[, deps], fn)

task方法用于定义具体的任务，其参数如表5-3所示。

表5-3　gulp.task的参数

参　　数	描　　述
name	任务名称
deps	可选参数。deps是一个包含任务列表的数组，是当前任务的依赖任务。task方法会并发执行这些依赖任务，等这些依赖任务完成后再执行当前任务
fn	任务函数，完成当前任务的操作

例如：

```
gulp.task('js', ['jscs', 'jshint'], function() {
    return gulp
        .src('./src/**/*.js')
```

① gulp.js. gulp/docs/FAQ.md[2016]. [2016]. https://github.com/gulpjs/gulp/blob/master/docs/FAQ.md.

```
            .pipe(concat('all.js'))
            .pipe(uglify())
            .pipe(gulp.dest('./build/'));
    });
```

以上示例定义了任务js，它依赖两个任务：jscs和jshint。只有当jscs和jshint任务完成后， js任务才能够执行。另外，jscs和jshint是并发执行，无法假设哪个先开始或结束。

如果一个任务的名字为default，就表明它是“默认任务”，在命令行直接输入gulp命令，就可运行该默认任务。

gulp支持异步任务。只要任务函数fn满足以下条件之一，那么这个任务就可以异步执行：

（1）接受一个回调函数参数。示例代码如下：

```
// run a command in a shell
var exec = require('child_process').exec;
gulp.task('jekyll', function(cb) {
  // build Jekyll
  exec('jekyll build', function(err) {
    if (err) return cb(err); // return error
    cb(); // finished task
  });
});
```

（2）返回一个stream对象。示例代码如下：

```
gulp.task('somename', function() {
  var stream = gulp.src('client/**/*.js')
    .pipe(minify())
    .pipe(gulp.dest('build'));
  return stream;
});
```

（3）返回一个promise对象。示例代码如下：

```
var Q = require('q');

gulp.task('somename', function() {

  var deferred = Q.defer();

  // do async stuff

  setTimeout(function() {

    deferred.resolve();

  }, 1);

  return deferred.promise;

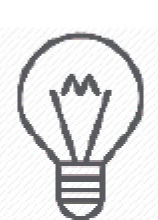
});
```

2. gulp.src(globs[, options])

src方法用于产生数据流。它输出符合匹配模式的文件，将文件转换成（返回）一个Vinyl files①的stream对象，并且利用管道和其他任务连接起来，如表5-4所示。

表5-4　gulp.src的参数

参数	描　　述
globs	glob匹配模式字符串或者匹配模式数组，用于指定所要处理的文件，其语法是node-glob②语法（除了negation模式）
options	可选参数。options是一个配置对象。它支持node-glob和glob-stream③的所有选项（除了ignore），并新增3个字段，如表5-5所示



常见的glob匹配模式有：

- src/app.js：指定确切的文件名src/app.js。
- src/*.js：src目录内后缀名为js的文件。
- src/**/*.js：src目录及其所有子目录中后缀名为js的文件。
- !js/app.js：除了js/app.js以外的所有文件。

表5-5　gulp.src options对象的字段

选　　项	类型	默　认　值	描　　述
options.buffer	布尔值	true	如果该项被设置为false，那么将会以stream方式返回file.contents而不是以文件buffer的形式。这在处理一些大文件的时候将会很有用
options.read	布尔值	true	如果该项被设置为false，那么file.contents会返回空值（null），也就是并不会去读取文件
options.base	字符串	glob的基准路径（glob2base④）。	指定glob的基准路径。例如client/js/**/*.js默认的基准路径是client/js，但是用户可以将其指定为client

① gulp.js. Vinyl adapter for the file system[OL]. [2016]. https://github.com/gulpjs/vinyl-fs.

② Isaac Z. Schlueter. glob functionality for node.js[OL]. [2016]. https://github.com/isaacs/node-glob.

③ gulp.js. File system globs as a stream[OL]. [2016]. https://github.com/gulpjs/glob-stream.

④ Eric Schoffstall. Extracts a base path from a node-glob instance[OL]. [2016]. https://github.com/contra/glob2base.

下面举例说明options.base的用法。

假如在client/js/somedir目录中，有一个文件somefile.js，options.base选项的用法举例如下：

```
// 匹配 'client/js/somedir/somefile.js'
// 并且将 'base' 解析为 'client/js/'
gulp.src('client/js/**/*.js')
  .pipe(minify())
// 写入 'build/somedir/somefile.js'
  .pipe(gulp.dest('build'));

gulp.src('client/js/**/*.js', { base: 'client' })
  .pipe(minify())
// 写入 'build/js/somedir/somefile.js'
  .pipe(gulp.dest('build'));
```

3. gulp.dest(path[, options])

dest方法把管道的输出写入文件，同时继续输出，这样开发人员可以通过多次调用dest方法，将输出写入到多个目录。如果目标目录不存在，dest方法会新建目录。dest方法的参数如表5-6所示。

表5-6　gulp.dest的参数

参数	描　述
path	文件输出目录。可以是字符串，也可以是一个返回路径的函数
options	可选参数。options是一个配置对象，它有两个选项： options.cwd：用于指定写入路径的基准目录，默认是当前目录 options.mode：用于指定写入文件的权限，默认是0777

4.gulp.watch(glob[, opts], tasks)或gulp.watch(glob[, opts, cb])

watch方法用于指定需要监视的文件。一旦这些文件发生变动，就会运行指定任务或回调函数。其参数如表5-7所示。

表5-7　gulp.watch的参数

参数	描　述
glob	glob匹配模式字符串或者匹配模式数组（同gulp.src），指定需要监视的文件
opts	可选参数。opts是一个配置对象，是传给gaze[①]的参数。gulp.watch使用gaze监视文件的变动

① Kyle Robinson Young. A globbing fs.watch wrapper built from the best parts of other fine watch libs[OL]. [2016]. https://github.com/shama/gaze.

（续表）

<table>
<tr><th>参数</th><th>描　　述</th></tr>
<tr><td>tasks</td><td>任务名称数组。文件变动后gulp.watch会执行这个数组指定的一个或多个任务。例如：<pre><code>var watcher = gulp.watch('js/**/*.js', ['uglify','reload']);</code></pre>上面的示例监视js目录及其子目录下所有后缀为js的文件。文件一旦有变动，就会执行uglify和reload任务</td></tr>
<tr><td>cb</td><td>回调函数。每次文件变动后gulp.watch会执行这个回调函数。这个回调函数cb(event)接受一个event对象参数，event对象使用以下两个字段描述文件的变动：
• event.type：字符串类型，指出发生变动的类型，例如added、changed、deleted或renamed
• event.path：字符串类型，指出触发该事件的文件路径
例如：<pre><code>gulp.watch('js/**/*.js', function(event) {
  console.log('File' + event.path + 'was' + event.type +', running tasks...');
});</code></pre></td></tr>
</table>

了解了这4个gulp API以后，下面读者可尝试编写一个default任务。该任务调用管道函数（.pipe）将各个操作，包括读取子目录下所有js文件，使用jshint进行代码分析，将所有js文件合并成all.js，压缩后写入build目录等操作串连起来。示例代码如下：

```
gulp.task('default', function () {
   return gulp
      .src('**/*.js')
      .pipe(jshint())
      .pipe(concat('all.js'))
      .pipe(uglify())
      .pipe(gulp.dest('./build/'));
});
```

5.5.3 运行gulp任务

要运行gulp任务，首先需要使用npm命令安装gulp。

1. 全局安装gulp命令行接口

全局安装gulp命令行接口的命令如下：

```
C:\jasmine-demo>npm install -g gulp
```

安装完成后运行以下命令以显示gulp的当前版本：

```
C:\jasmine-demo>gulp -v
[00:02:47] CLI version 3.9.1
```

2. 本地安装gulp

在本地安装gulp的命令如下：

```
C:\jasmine-demo>npm install gulp --save-dev
```

3. 定义gulp任务

在项目根目录创建gulp配置文件gulpfile.js，在这个文件里使用gulp API定义任务。示例代码如下：

```
var gulp = require('gulp');
gulp.task('hello-world', function () {
   console.log('Our first gulp task!');
});
```

Node.js使用require函数加载gulp软件包，获得gulp对象，然后定义任务hello-world。

4. 运行gulp任务

在命令控制台执行gulp <task> <othertask>命令，就可以运行指定gulp任务了。例如：

```
C:\jasmine-demo>gulp hello-world
[10:25:38] Using gulpfile C:\jasmine-demo\gulpfile.js
[10:25:38] Starting 'hello-world'…
Our first gulp task!
[10:25:38] Finished 'hello-world' after 435 μs
```

5.6　Karma和gulp集成

在gulp里定义Karma任务不需要额外的插件[1]，gulpfiles.js示例如下：

---

① Karma. Example of using Karma with Gulp[OL]. [2016]. https://github.com/karma-runner/gulp-karma#do-we-need-a-plugin.

```
var gulp = require('gulp');

var Server = require('karma').Server;


/**

 * Run test once and exit

 */

gulp.task('testonce', function (done) {

  new Server({

    configFile: __dirname + '/karma.conf.js',

    singleRun: true

  }, done).start();

});
```

以上示例在创建Server对象时传入一个配置对象，该配置对象可以覆盖karma.conf.js里的配置。此处仅设置两个字段：

- configFile：Karma配置文件路径。__dirname是Node.js的一个全局对象，代表当前执行代码所在的目录名。
- singleRun：设为持续集成模式。

在命令控制台运行以下命令执行任务testonce，就可以调用Karma，开始单元测试。

```
C:\jasmine-demo>gulp testonce
```

第6章 AngularJS应用的单元测试

AngularJS是一款来自Google的前端JavaScript框架，也是一个著名的SPA（single-page-application，单页应用）框架。它有如下一些特点：

- 完整的解决方案。很多前端开发功能需求被AngularJS原生支持，开发人员不需要依赖其他框架提供的方案。
- 容易学习。AngularJS使用HTML作为模板语言，通过扩展HTML的语法，让开发人员能够更清晰、简洁地开发应用组件。
- 社区活跃，可以很方便地找到文档和各种解决方案。
- 开源，同时得到Google的大力支持。
- 高可测试性。AngularJS将应用程序的测试看得跟应用程序的编写一样重要。

以上这些特性使得AngularJS迅速成为了JavaScript的主流框架。本章假设读者有一定的AngularJS编程经验，在此基础上介绍AngularJS 1.x应用的单元测试最佳实践。

本章将介绍：

- 测试AngularJS应用的挑战
- 初识ngMock
- AngularJS单元测试最佳实践

6.1 测试AngularJS应用的挑战

测试AngularJS应用和测试一般JavaScript应用相比，主要要面对以下几个挑战：

1. 引导AngularJS应用

当启动浏览器加载AngularJS应用时，除了利用<script>标签引用AngularJS的库文件以外，还需要通过加载应用的模块，编译DOM来初始化AngularJS应用。

初始化AngularJS应用有两种方案。一种方案是通过ng-app指令自动引导，加载应用模块（以下示例的optionalModuleName模块），代码如下：

```
<!DOCTYPE html>
<html ng-app="optionalModuleName">
  <body>
    …
    <script src="angular.js"></script>
  </body>
</html>
```

另一种方案是使用angular.bootstrap函数手动引导（加载以下示例中的myApp模块），代码如下：

```
<!DOCTYPE html>
<html>
  <body>
    ...
    <script src="angular.js"></script>
    <script>
    angular.element(function() {
      angular.bootstrap(document, ['myApp']);
    });     </script>
  </body>
</html>
```

但是，在单元测试时，开发人员怎样才能在不能使用ng-app和angular.bootstrap的情况下引导AngularJS应用？

2. 使用Controller

Controller在AngularJS应用里用到的地方很多，它的主要作用是向视图传递数据、操作页面逻辑和增强视图。开发人员可以通过ng-controller指令、Directive或者Router组件创建Controller的实例。

在单元测试时，如何创建并初始化一个 Controller实例并对它进行测试？

3. 使用Service

在AngularJS应用里，业务逻辑一般会被封装在Service中。Service（factory、service、constant、valuet和provider）对象都是singleton（单例）对象。AngularJS负责创建并初始化它们，并通过依赖注入提供给Controller、Directive或者其他Service。

在单元测试时，如何使用测试替身代替并隔离这些Service，以及控制它们的行为？

4. 使用Directive

AngularJS推荐使用Directive操作DOM。一个Directive包含HTML模板和扩展DOM行为的JavaScript代码。那么如何测试AngularJS Directive？

除了以上几点，还需要考虑如何测试$scope、$http和$log等AngularJS内建对象。本书将在后续章节一一回答。

6.2　初识ngMock

可测试性一开始就被作为AngularJS立项的核心设计目标之一。实际上，AngularJS的创建者Misko Hevery当初在Google的工作就是测试顾问。

AngularJS同时支持单元测试和端到端测试。端到端测试可以基于Protractor，这是AngularJS团队创建的测试库（本书第11章会详细介绍Protractor）。AngularJS的单元测试是通过内建的模块ngMock来完成的。ngMock包含了一系列帮助开发人员测试 AngularJS应用程序的工具和方法，它有两个主要函数，如表6-1所示。

表6-1　ngMock的主要函数

函　　数	描　　述
angular.mock.module	在单元测试中加载模块
angular.mock.inject	在单元测试中注入依赖组件

除了这两个函数以外，ngMock另外提供了一些组件协助完成AngularJS应用程序的测试，下面依次介绍这些功能。

6.2.1　准备测试环境

新建一个项目目录C:\ngmock-demo，在该目录下执行npm init命令创建package.json。该文件中的devDependencies和dependencies字段内容如下：

```
"devDependencies": {
  "jasmine-core": "^2.5.2",
  "karma": "^1.3.0",
  "karma-chrome-launcher": "^2.0.0",
  "karma-jasmine": "^1.0.2"
},
"dependencies": {
  "angular": "^1.5.9",
  "angular-mocks": "^1.5.9",
  "jquery": "^3.1.1"
}
```

其中angular-mocks就是ngMock软件包。接下来执行npm install命令安装以上所有这些软件包：

```
C:\ngmock-demo>npm install
```

在项目目录下执行karma init命令创建karma.conf.js（本书使用Karma来驱动单元测试）。配置文件中的files字段内容如下：

```
files: [
  'node_modules/jquery/dist/jquery.js',
  'node_modules/angular/angular.js',
  'node_modules/angular-mocks/angular-mocks.js',
  'app/**/*.js',
],
```

将测试所需的类库、源文件和测试用例都添加到files字段。其中angular-mocks.js是ngMock的类库。

6.2.2 理解模块（Module）

大多数应用程序都有个 main 函数来初始化、连接以及启动整个应用，但是AngularJS

使用模块化的组织方式，没有main函数。开发人员可以把AngularJS的模块想象成一个容器，包括Controller、Service、Filter、Directive等组件，如图6-1所示。

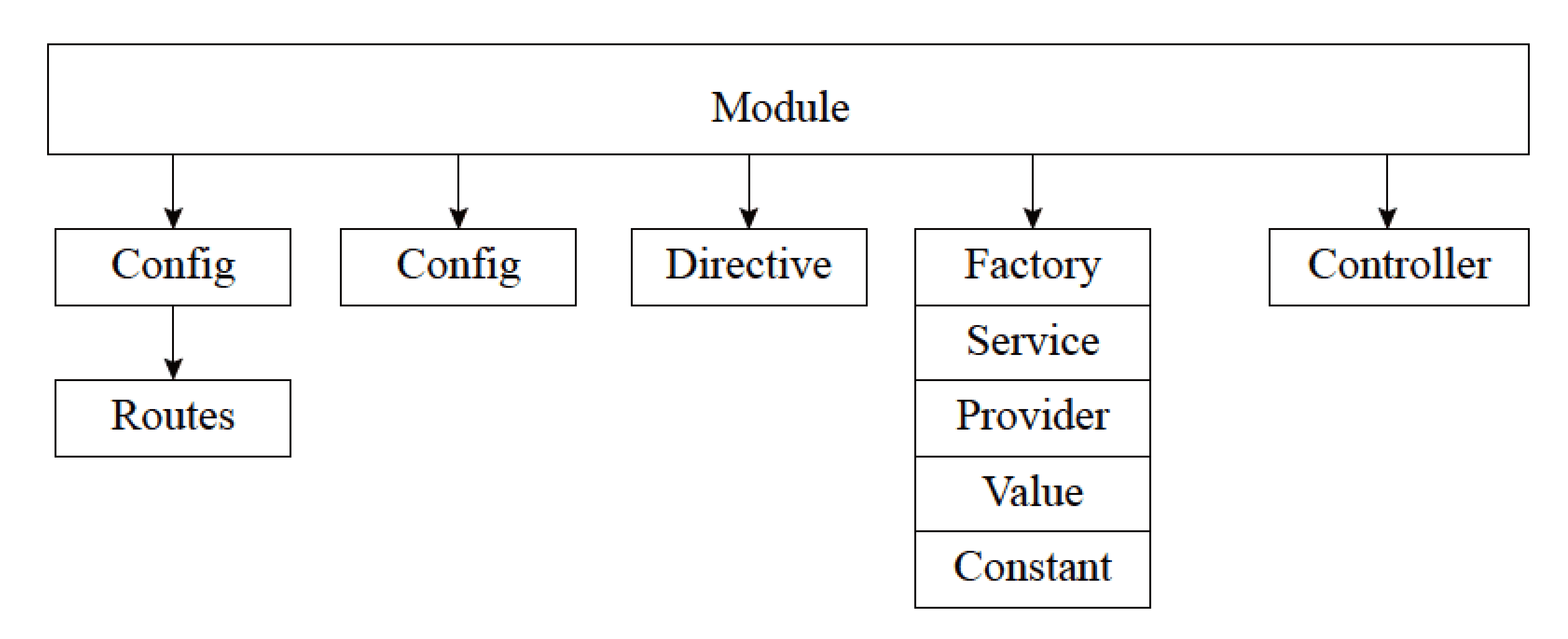


图6-1 AngularJS模块

在AngularJS应用程序里，开发人员使用以下方式创建和获取模块。

- 创建一个模块

创建模块的代码如下：

```
// Setter
angular.module('demoApp', ['helperModule']);
```

- 获取一个模块

获取模块的代码如下：

```
// Getter
angular.module('demoApp');
```

AngularJS应用通过ng-app命令或angular.bootstrap函数声明用哪个模块来引导程序。和main函数相比，模块化方式非常有利于单元测试的代码书写。因为使用模块化方式，在单元测试中只需要加载需要测试的特定模块，而不是所有的模块。

ngMock的angular.mock.module函数用于在单元测试中加载模块。它接受3种参数：

- 字符串：已有模块的名字。
- 回调函数：创建一个新的匿名模块。
- 对象：创建一个新的匿名模块。

接下来分别介绍这3种参数。

1. 字符串参数

假设AngularJS应用定义了两个模块，并在这两个模块里定义了一系列Controller、

Service等组件，如下所示：

```
angular.module('demoApp.module', []);
angular.module('helperModule', []);
```

如果要在单元测试里测试这些组件，必须先加载相应的模块。angular.mock.module字符串参数指定了需要加载的模块。以下的测试用例用于加载demoApp.module：

```
describe('angular.mock.module', function () {
    it('should load module with string alias', function () {
        angular.mock.module('demoApp.module');
        expect(true).toBe(true);
    });
});
```

可以同时加载demoApp.module和helperModule，代码如下：

```
angular.mock.module('demoApp.module', 'helperModule');
```

也可以分别加载，代码如下：

```
angular.mock.module('demoApp.module');
angular.mock.module('helperModule');
```

2. 回调函数

为了达到单元测试的隔离目的，有时候需要创建一些临时组件来代替实际的组件。创建这些临时组件需要一个容器（模块）。当angular.mock.module使用回调函数时，它会定义一个匿名模块，回调函数里声明的组件都被注册在匿名模块里。例如以下示例代码在匿名模块里创建了一个constant组件：

```
it('should load module with anonymous function', function () {
    angular.mock.module(function($provide) {
        $provide.constant('apiUrl', 'http://www.example.com/api');
        // We could register other provider services here…e.g.
        // $provide.value('apiKey', 'apisecret');
```

```
    });
    expect(true).toBe(true);
});
```

3. 对象参数

当使用对象参数时，angular.mock.module也会定义一个匿名模块，这个对象参数里的每一对字段/值将成为一个Value组件。例如以下示例在匿名模块里创建了两个Value组件：apiKey和basicService：

```
it('should load module with object', function () {
    angular.mock.module({
        'apiKey': 'apisecret',
        'basicService': {
            changeMessage: function (msg) {
                return msg + '!!!';
            }
        }
    });
    expect(true).toBe(true);
});
```

对象参数和回调函数都会定义一个匿名模块并在匿名模块中创建测试用的临时组件，它们的区别是：对象参数只能创建Value组件，而回调函数可以创建其他类型的组件。



AngularJS内建的ng和ngMock模块会被自动加载，而不需要在单元测试里特地调用angular.mock.module来加载这两个模块。

6.2.3　理解注入机制（Inject）

为了测试某些组件，首先利用angular.mock.module加载模块，获得模块中各个组件的实例（参见图6-2）后才能对这些实例进行单元测试。那么如何在单元测试中得到组件的实例呢？

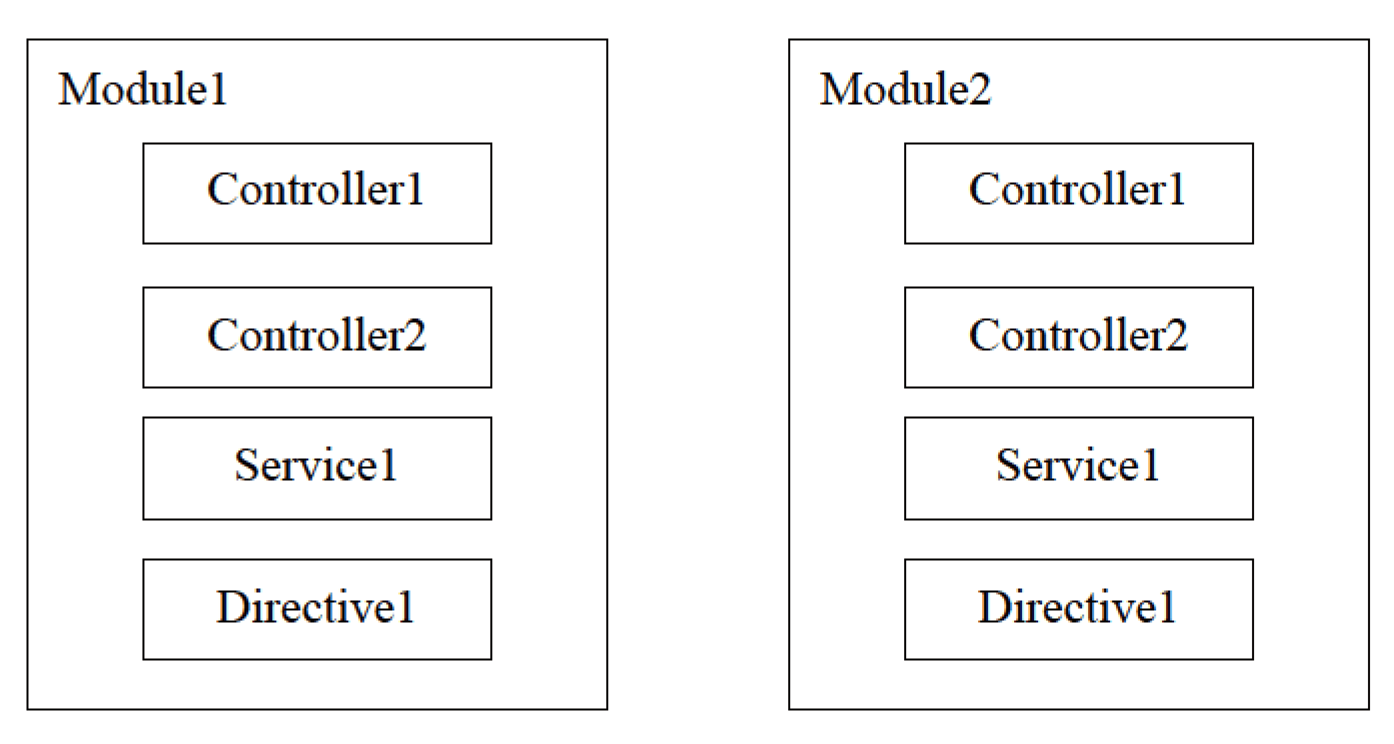


图6-2　模块和组件

在AngularJS应用中，如果需要一个被依赖的组件实例，则不会调用new操作符，而是利用AngularJS内建的依赖注入机制，由AngularJS创建并注入到应用中。依赖注入机制使得各个组件之间耦合度低，可测试性好，因此开发人员在单元测试中可以方便地对依赖组件进行替换。

AngularJS的依赖注入机制是通过$injector完成的。每一个AngularJS应用都有一个$injector管理依赖查询以及实例化组件。事实上，使用ng-app指令引导AngularJS应用时，ng-app除了加载相应模块以外，同时还创建一个$injector，该$injector负责创建并管理AngularJS组件的所有实例，包括Directive和Controller等，如图6-3所示。

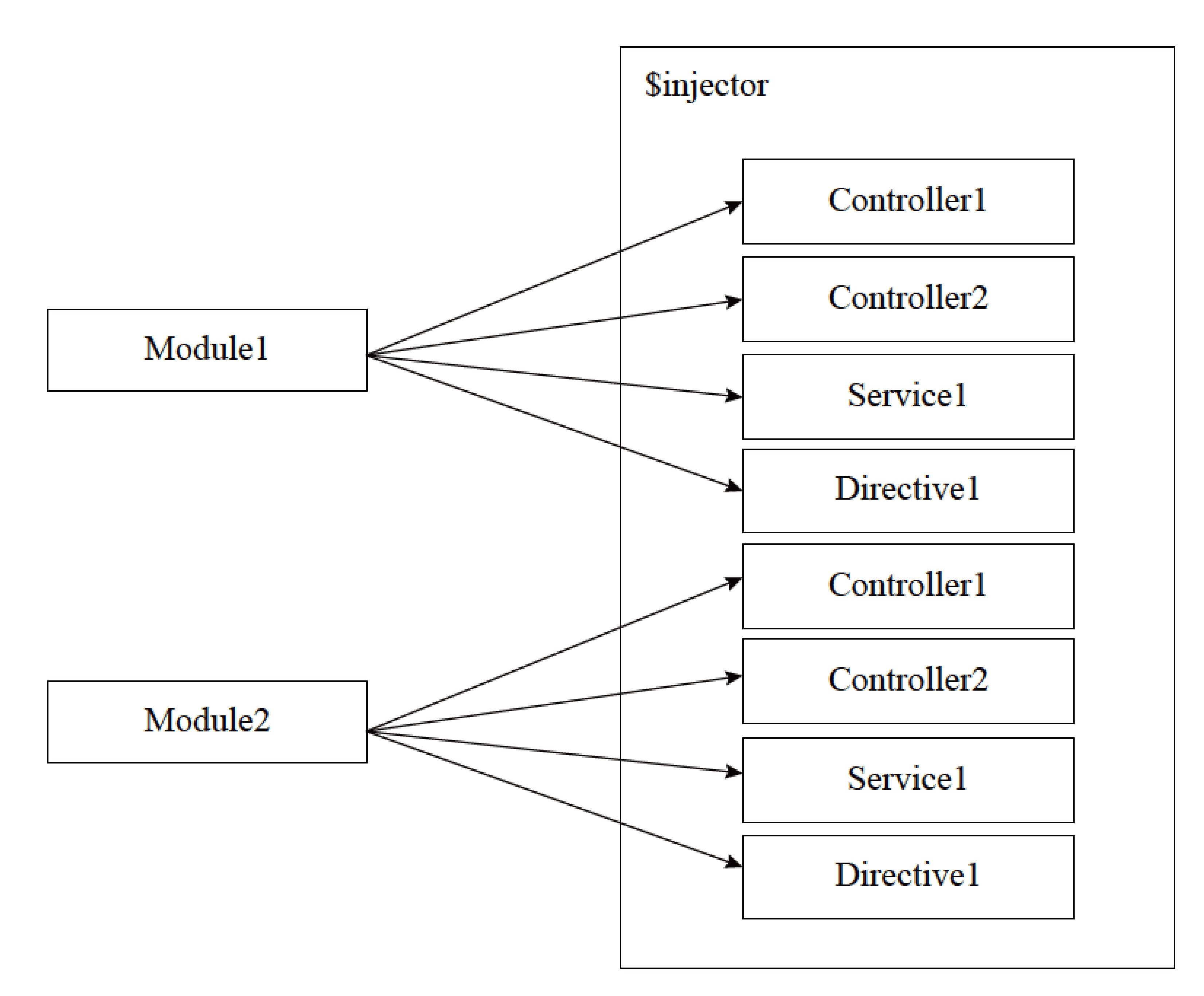
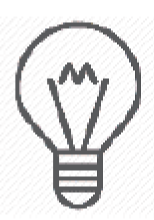


图6-3　AngularJS $injector



Module不是Namespace。

虽然AngularJS的模块是容器，包含Controller、Directive等组件，但是它没有提供任何命名空间的功能。在一个应用里，所有模块里的组件都会被一个$injector管理。如果不同模块里有同名的组件，那么这些同名组件（参见图6-3）就会有命名冲突，后加载的组件会代替前面的组件。事实上在AngularJS单元测试中，开发人员可以利用这个特性定义一些同名的“假”组件（测试替身）来代替实际组件，起到隔离的效果。

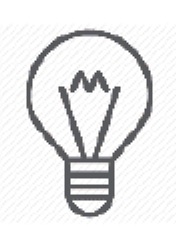
在单元测试时开发人员也需要一个$injector来管理并实例化依赖组件。ngMock的angular.mock.inject函数为每个测试用例创建并封装一个$injector，注入测试所需要的组件实例。

为了理解angular.mock.inject函数，需要准备一个被测试的Service（C:\ngmock-demo\app\basic\product.service.js）。

product.service.js文件的内容如下：

```
/* product.service.js */
(function() {
   'use strict';
   angular.module('demoApp.basic', []);
   angular
      .module('demoApp.basic')
      .service('ProductService', ProductService);
   function ProductService() {
      return function () {
         return [{name: 'foo'}, {name: 'bar'}];
      };
   }
})();
```

以上示例中，模块demoApp.basic里定义一个简单的Service（ProductService），这个Service其实是一个函数，调用它会返回一个对象数组。



注意：通常定义模块的代码会放到单独的app.js里，考虑篇幅所限，这里将所有功能代码放到一个文件里。本章AngularJS代码基本遵循John Papa的AngularJS样式指南[①]。

接下来编写单元测试代码C:\ngmock-demo\app\basic\product.service.spec.js，内容如下：

```
/* product.service.spec.js */
describe('ProductService ', function () {
  var ProductService;
  beforeEach(function () {
    angular.mock.module('demoApp.basic');
    // Get the service from the injector
    angular.mock.inject(function (_ProductService_) {
      ProductService = _ProductService_;
    });
  });
  it('should retrieve products successfully', function () {
    var result = ProductService();
    expect(result).toEqual([{name: 'foo'}, {name: 'bar'}]);
  });
});
```

此示例依次完成以下工作：

（1）定义一个变量ProductService，准备保存ProductService实例。

（2）使用angular.mock.module函数加载demoApp.basic。

（3）调用angular.mock.inject函数，传入一个回调函数，参数为_ProductService_。angular.mock.inject创建$injector，并且在已经加载的模块里寻找名为ProductService的组件。（注意参数_ProductService_是AngularJS社区的一种使用惯例，通过前后下画线包装需要注入的组件，$injector在解析时会自动去除两端的下画线。实际上在注入组件名前后添加下画线也便于编写测试，使得测试用例里内局部变量ProductService和注入组件名一致）

（4）$injector在demoApp.basic模块里找到ProductService组件，将它的实例通过_

① John Papa. Angular 1 Style Guide[OL]. [2016]. https://github.com/johnpapa/angular-styleguide/tree/master/a1.

ProductService_参数传入函数，然后赋给变量ProductService。

完成以上工作后，即可在测试用例里使用ProductService实例。

在项目根目录下执行karma start命令，测试通过。然后对原先的JavaScript代码稍作修改，增加一个新的HelperService，并且让ProductService依赖于HelperService组件，代码如下：

```
angular
    .module('demoApp.basic')
    .service('ProductService', ProductService)
    .service('HelperService', HelperService);
ProductService.$inject = ['HelperService'];
function ProductService(HelperService) {
    return function() {
        return HelperService();
    };
}
function HelperService() {
    return function () {
        return [{name: 'foo'}, {name: 'bar'}];
    };
}
```

在不修改测试用例的情况下此单元测试仍然可以通过，因为angular.mock.inject函数不仅能解析ProductService，而且能自动解析ProductService所依赖的组件HelperService。

上面的测试用例有个缺陷，就是同时测试了ProductService和HelperService。单元测试应该是无依赖和隔离的。测试ProductService时开发人员需要将它所依赖的HelperService隔离，用“假”的HelperService代替实际的HelperService。为此修改product.service.spec.js，代码如下：

```
/* product.service.spec.js */
describe('ProductService ', function () {
  var ProductService;
  beforeEach(function () {
```

```
    angular.mock.module('demoApp.basic');
    angular.mock.module({
      'HelperService': function () {
        return [{name: 'baz'}, {name: 'qux'}];
      }
    });
    // Get the service from the injector
    angular.mock.inject(function (_ProductService_) {
      ProductService = _ProductService_;
    });
  });
  it('should retrieve products successfully', function () {
    var result = ProductService();
    expect(result).toEqual([{name: 'baz'}, {name: 'qux'}]);
  });
});
```

以上示例调用angular.mock.module定义一个匿名模块，并且利用对象参数创建一个临时Value组件HelperService，准备取代应用程序里的HelperService。同时修改断言的期望值（[{name: 'baz'}, {name: 'qux'}]）以满足“假”的HelperService。测试结果显示使用了匿名模块里的HelperService，而不是demoApp.basic里的HelperService。

以上单元测试最终使用匿名模块的HelperService取代应用程序里的HelperService，其原因是：虽然有多个模块，但是测试程序只有一个$injector，各个模块里的所有组件最终都由这个$injector管理，所以晚加载的匿名模块里的同名HelperService会覆盖demoApp.basic里的HelperService，如图6-4所示。

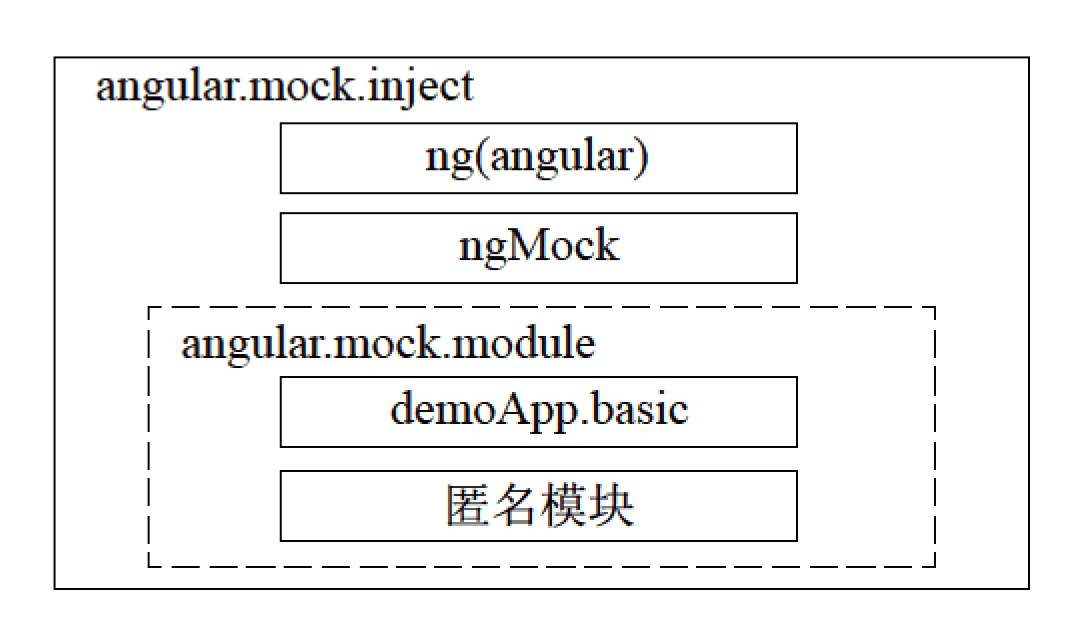
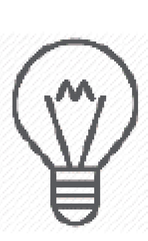


图6-4　单元测试中的$injector

需要注意的是，在测试代码里模块加载的顺序非常重要。如果像以下代码里先定义匿名模块，然后加载demoApp.basic，那么测试用例最终使用的仍是实际的HelperService，在单元测试中需要避免这种情况：

```
angular.mock.module({
  'HelperService': function() {
    return [{name: 'baz'}, {name: 'qux'}];
  }
});
angular.mock.module('demoApp.basic');
```



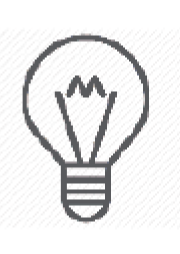
AngularJS内建的ng和ngMock模块会被$injector自动加载，并且会加载在它的模块列表的最前面。查看angular-mocks.js中angular.mock.inject函数的实现，代码如下：

```
window.inject = angular.mock.inject = function() {
    …
    ////////////////////
    function WorkFn() {
      var modules = currentSpec.$modules || [];
      …
      modules.unshift('ngMock');
      modules.unshift('ng');
```

beforeEach里的angular.mock.inject函数会为每个测试用例创建一个$injector。如果多个测试用例要共享一个$injector，那么可以在beforeAll里创建它，但是必须额外调用angular.mock.module.sharedInjector函数，如下所示：

```
describe('Shared Injector', function () {
  var ProductService;
  angular.mock.module.sharedInjector();
  beforeAll(angular.mock.inject(function (_ProductService_) {
```

```
    ProductService = _ProductService_;
  }));
});
```



angular.mock.module和angular.mock.inject函数都被发布到全局的window对象处，所以可以直接使用module()和inject()。本书为了避免混淆，还是使用angular.mock.module和angular.mock.inject函数。

6.3 AngularJS单元测试最佳实践

6.3.1 测试Controller

AngularJS的Controller组件负责向视图传递数据、控制页面逻辑，是AngularJS里广泛用到的一个组件。

1. 准备Controller示例

定义一个Controller（C:\ngmock-demo\app\controller\products.controller.js），代码如下：

```
/* products.controller.js */
(function() {
   'use strict';
   angular.module('demoApp.controller', ['demoApp.basic']);
   angular.module('demoApp.controller')
      .controller('ProductsController', ProductsController)
   ProductsController.$inject = ['ProductService'];
   function ProductsController(ProductService) {
      var vm = this;
      vm.products = ProductService();
   }
})();
```

ProductsController被定义在模块demoApp.controller里，它依赖上一节定义的demoApp.basic模块里的ProductService。在ProductsController的构造函数里，从ProductService取得的数据被赋给vm.products。

2. $controller

要测试ProductsController，必须在单元测试中获得它的实例。但是angular.mock.inject不能直接实例化Controller，为此ngMock提供了一个特殊的$controller服务，可以通过angular.mock.inject获得$controller的实例，代码如下：

```
var $controller;

beforeEach(function() {

   angular.mock.inject(function(_$controller_) {

      $controller = _$controller_;

   });

});
```

$controller其实是一个函数，获得$controller的实例后就可以调用它来得到被测应用程序的Controller实例。$controller函数接受3个参数：

```
$controller(constructor, locals, [bindings]);
```

- constructor：字符串或者是一个回调函数。字符串指的是要获取的Controller的名称。回调函数用来创建一个匿名Controller。
- locals：对象。其字段名和传入Controller构造函数的参数名匹配，例如ProductsController构造函数的ProductService参数。可以通过该对象传入“假”的依赖组件。
- bindings：可选参数，也是一个对象。该对象的各个字段（属性或方法）会被自动绑定到Controller实例上。当被测应用程序的Controller使用controllerAs语法时，测试用例代码可以通过bindings参数对Controller进行初始化。

3. 通过名称获得Controller实例

利用Controller名称获得相应的实例是最常见的一种情况。以下是完整的测试代码products.controller.spec.js：

```
/* products.controller.spec.js */
```

```
describe('ProductsController', function() {
   var $controller;
   beforeEach(function() {
      angular.mock.module('demoApp.controller');
      angular.mock.inject(function(_$controller_) {
         $controller = _$controller_;
      });
   });
   it('should return products list', function () {
      var productsController;
      productsController = $controller('ProductsController',{});
      expect(productsController.products).toEqual(
         [{ name: 'foo' }, { name: 'bar' }]);
   });
});
```

以上示例代码中，demoApp.controller模块被加载后（同时也加载了依赖模块demoApp.basic），测试用例调用$controller函数，传入'ProductsController'字符串，从而获得ProductsController的实例，然后就可以对ProductsController进行测试。

4. 通过回调函数创建匿名Controller

$controller函数的constructor参数也可以是一个回调函数，创建一个匿名Controller，通常用来帮助开发人员做些原型设计，而不是测试应用程序原有的Controller。以下示例使用了回调函数：

```
it('should return products list by anonymous controller', function () {
   var productsController;
   productsController = $controller(function (ProductService) {
      var vm = this;
      vm.products = ProductService();
   }, {});
   expect(productsController.products).toEqual(
```

```
        [{ name: 'foo' }, { name: 'bar' }]);
});
```

5. 使用locals注入Controller的依赖组件

ProductsController依赖ProductService，上面的测试代码测试ProductsController的同时也使用了应用程序真实的ProductService。如果想要在单元测试时隔离Controller，可以使用$controller函数的locals参数传入“假”的依赖组件。以下示例“假”的ProductService（mockService）通过locals对象传入$controller函数中：

```
it('should return products list by mock service', function () {
    var productsController;
    var mockService = function () {
        return [{ name: 'baz' }, { name: 'qux' }];
    };
    productsController = $controller('ProductsController',
        { ProductService: mockService });
    expect(productsController.products).toEqual(
        [{ name: 'baz' }, { name: 'qux' }]);
});
```

6. 使用bindings初始化Controller

$controller函数的bindings对象参数的各个字段（属性或方法）会被自动绑定到Controller实例上，通常用于对Controller实例进行初始化。示例代码如下：

```
it('should initialize controller by bindings', function () {
    var productsController;
    var bindings = [{ name: 'baz' }, { name: 'qux' }];
    productsController = $controller('ProductsController',
        {}, {data: bindings});
    expect(productsController.data).toEqual(
        [{ name: 'baz' }, { name: 'qux' }]);
});
```

以上示例中，注意传入$controller的对象{data: bindings}，其字段data会被自动绑定到ProductsController实例上；最后验证productsController的data属性。

6.3.2 单元测试中的Scope

Controller向视图传递数据有两种方式：controllerAs和$scope。上一节示例里的ProductsController使用了controllerAs语法，但是还有大量开发人员习惯使用$scope（参见图6-5）。那么在单元测试中如何处理$scope呢？



图6-5　$scope关联Controller和视图

将ProductsController略作修改，创建一个使用$scope方式的ProductsWithScopeController（C:\ngmock-demo\app\scope\productswithscope.controller.js）

```
/* productswithscope.controller.js */
(function() {
   'use strict';
   angular.module('demoApp.rootscope', ['demoApp.basic']);
   angular.module('demoApp.rootscope')
      .controller('ProductsWithScopeController',ProductsWithScopeController)
   ProductsWithScopeController.$inject = ['$scope', 'ProductService'];
   function ProductsWithScopeController($scope, ProductService) {
      $scope.products = ProductService();
   }
})();
```

ProductsWithScopeController依赖$scope和ProductService。$scope本身是一个对象，如果仅仅作为一个数据模型在Controller和视图间传递数据，那么在单元测试时可以使用一个空对象代替实际的$scope对象。示例代码如下：

```
   it('should return products list', function () {
      var $scope = {};
```

```
        $controller('ProductsWithScopeController', {$scope: $scope});
        expect($scope.products).toEqual(
            [{ name: 'foo' }, { name: 'bar' }]);
    });
```

但是AngularJS实际的$scope要复杂得多。每个AngularJS应用都有一个$rootScope，这是一个顶层的Scope对象，开发人员可以通过它的$new方法来创建新的$scope。这些$scope对象都被包含在$rootScope里。$scope之间可以嵌套，它们有继承或独立的关系，如图6-6所示。除了可以作为数据模型以外，$scope还有自己的属性和方法（例如$apply如$on）。

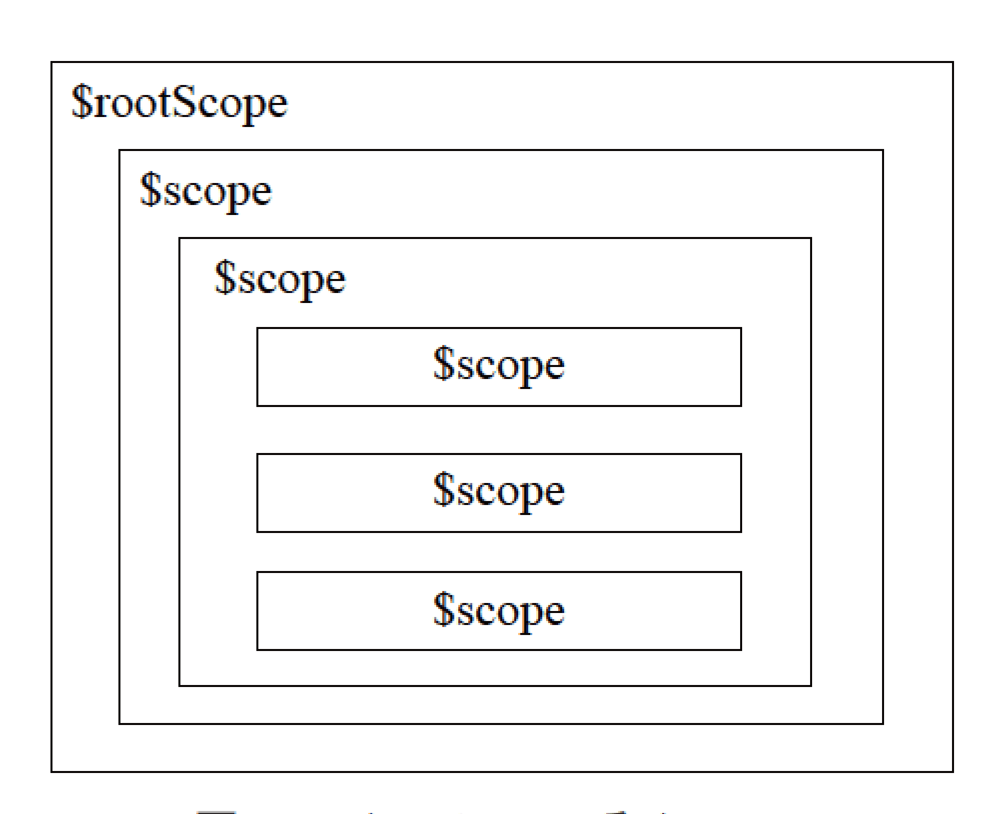


图6-6　$rootScope和$scope

为了在单元测试中模拟$scope，ngMock提供了对应的测试专用$rootScope，可以利用它在单元测试中创建$scope。示例代码如下：

```
/* productswithscope.controller.spec.js */
describe('ProductsWithScopeController', function() {
    var $controller;
    var $rootScope;
    beforeEach(function () {
        angular.mock.module('demoApp.rootscope');
        angular.mock.inject(function (_$controller_, _$rootScope_) {
            $controller = _$controller_;
            $rootScope = _$rootScope_;
        });
```

```
    });
    it('should return products list', function () {
        var $scope = $rootScope.$new();
        $controller('ProductsWithScopeController', {$scope: $scope});
        expect($scope.products).toEqual(
            [{ name: 'foo' }, { name: 'bar' }]);
    });
});
```

以上代码所创建的Scope可以继承以下内容：

```
    it('should return message from parent scope', function () {
        var $scope = $rootScope.$new();
        var $childScope = $scope.$new();
        $scope.message = 'Parent Scope';
        $controller('ProductsWithScopeController', {$scope: $childScope});
        expect($childScope.products).toEqual(
            [{ name: 'foo' }, { name: 'bar' }]);
        expect($childScope.message).toEqual('Parent Scope');
    });
```

ngMock测试专用$rootScope除了具有AngularJS内建的$rootScope的功能以外，还额外提供以下两个方法以协助测试，如表6-3所示。

表6-3　ngMock的$rootScope方法

方　法	描　述
$countChildScopes()	当前Scope包含的所有直接和间接Scope的数量
$countWatchers()	当前Scope包含的所有直接和间接Scope里的Watcher数量

后面章节测试AngularJS其他组件的时候会介绍$rootScope的更多用法。

6.3.3　测试HTTP交互

运行在浏览器里的JavaScript代码使用Ajax和服务器进行交互已经是现代Web应用中

不可或缺的一环。$http是AngularJS提供的一个和服务器进行HTTP交互的核心组件。它的底层通过$httpBackendProvider，利用浏览器的XMLHttpRequest或JSONP与服务器进行连接，如图6-7所示。下面介绍使用$http的AngularJS应用程序进行单元测试的方法。

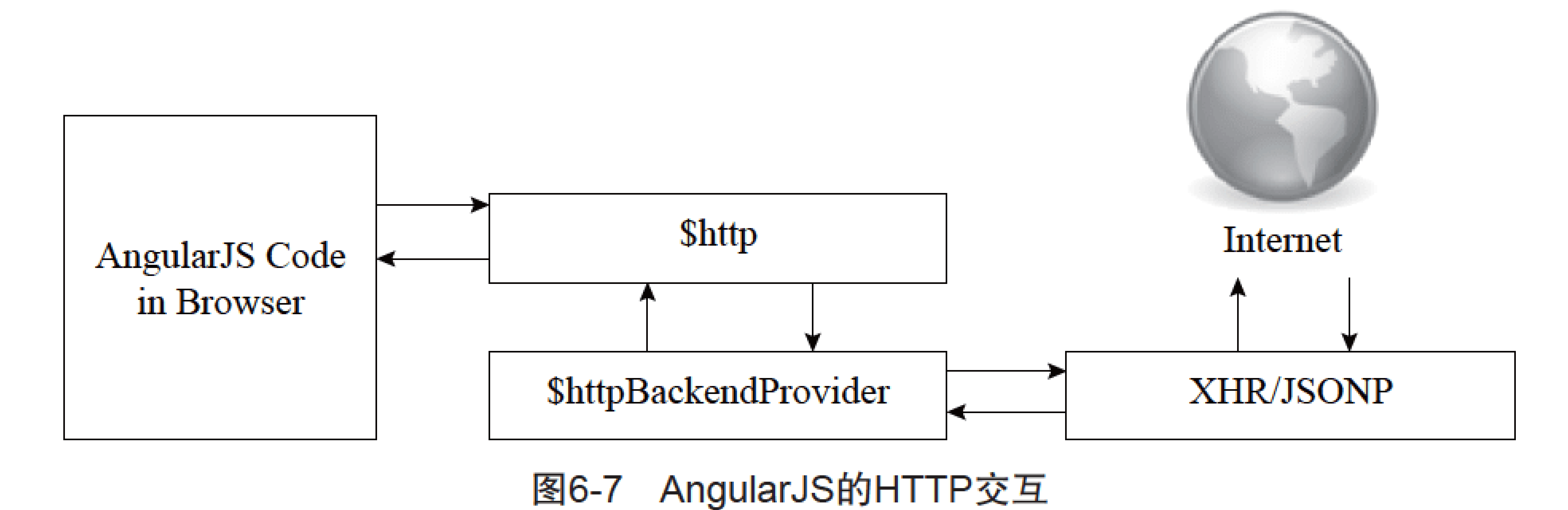


图6-7 AngularJS的HTTP交互

1. 准备$http示例

定义一个Factory（C:\ngmock-demo\app\http\basicHttp.factory.js），代码如下：

```
angular.module('demoApp.http', []);

angular

   .module('demoApp.http')

   .factory('basicHttpFactory', basicHttpFactory)

basicHttpFactory.$inject = ['$http'];

function basicHttpFactory($http) {

   return {

      getProductName: getProductName

   };

   function getProductName(url) {

      return $http.get(url)

         .then(function (result) {

            return result.data.name;

         });

   }

}
```

以上代码里basicHttpFactory调用$http从服务器获取某个产品的名字。



AngularJS中使用Service（Factory和Value等）来组织那些可被复用的业务逻辑。建议尽量将Ajax请求放到Service中去实现，而不要在Controller或Directive中直接注入$http。这样在单元测试中可以让“假”的Service返回预期结果，提高应用程序的可测试性。

2. 使用$httpBackend进行单元测试

$http会发送HTTP请求到服务器，但单元测试中需要测试代码能快速运行，及时反馈并且没有外部依赖，所以不希望HTTP请求被真正发送到实际服务器，而只是验证HTTP请求是否已经发送，并且能得到预定义好的数据。

ngMock提供了测试$http用的伪后端$httpBackend，如图6-8所示。

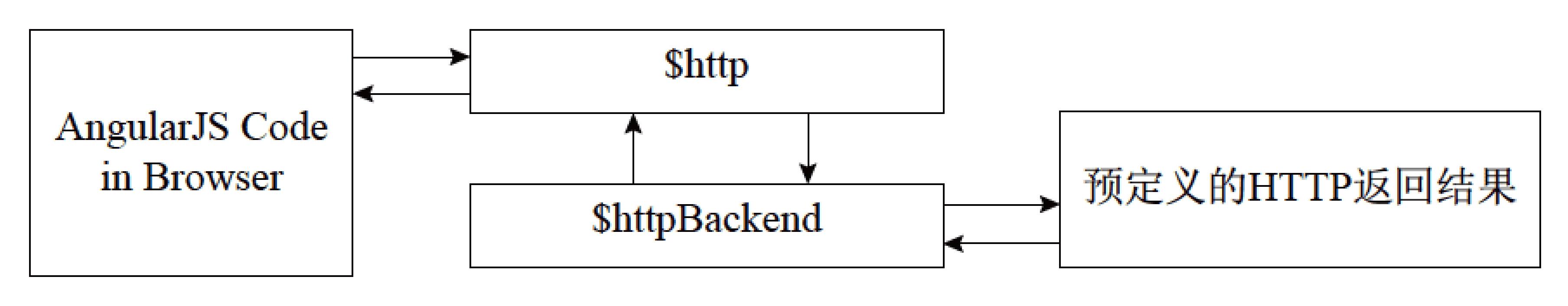


图6-8　使用$httpBackend

以下示例代码使用$httpBackend对basicHttpFactory的getProductName方法进行测试：

```
describe('basicHttpFactory', function () {
  var basicHttpFactory, $httpBackend;
  beforeEach(function () {
    angular.mock.module('demoApp.http');
    angular.mock.inject(function(_basicHttpFactory_, _$httpBackend_) {
      basicHttpFactory = _basicHttpFactory_;
      $httpBackend = _$httpBackend_;
    });
  });
  it('getProductName should get mocked data successfully', function () {
    var result;
    var url = 'http://localhost/foo/productinfo.json';
    $httpBackend
      .when('GET', url)
      .respond(200, { name: 'foo' });
    var promise = basicHttpFactory.getProductName(url);
```

```
        promise.then(function (data) {
            result = data;
        });
        $httpBackend.flush();
        expect(result).toEqual('foo');
    });
});
```

以上示例要关注以下4点：

（1）在测试用例里通过$httpBackend.when设置伪后端的响应条件和结果。一旦$http的请求匹配响应条件（GET http://localhost/foo/productinfo.json），那么伪后端就会返回成功状态码200以及数据。

（2）调用basicHttpFactory.getProductName方法发送HTTP请求。

（3）因为$http是异步的，getProductName方法返回一个Promise，所以需要设置Promise的回调函数。此时伪后端还没有响应HTTP请求。

（4）调用$httpBackend.flush函数让伪后端立即响应等待的HTTP请求，然后验证结果。

除了$httpBackend.when，开发人员还可以使用$httpBackend.expect设置请求的响应条件和结果。例如：

```
        $httpBackend.expect('GET', url)
            .respond(200, { name: 'foo' });
        expect($httpBackend.flush).not.toThrow();
```

$httpBackend.expect期待应用程序发出GET http://localhost/foo/productinfo.json请求。如果这个期待的请求不出现，$httpBackend.flush函数就会抛出异常。

3. when和expect对比

虽然$httpBackend.when和$httpBackend.expect都可以用来设置响应条件和结果，但是它们有很大区别，如表6-2所示。

表6-2　$httpBackend.when和$httpBackend.expect对比

$httpBackend.when	$httpBackend.expect
语法 .when(…).respond(…)	语法 .expect(…).respond(…)
新建一个后端定义（backend definition）。当应用程序请求符合条件时，伪后端返回预先定义的数据结果，但是伪后端不会对请求进行断言，这意味着不管有没有请求符合条件，都不会影响最终测试结果	新建一个请求期望（request expectation）。它会对应用程序的请求进行断言，并返回指定结果。如果预期的请求没有出现或者顺序不对，那么测试失败（$httpBackend.flush抛出异常）

（续表）

$httpBackend.when	$httpBackend.expect
黑盒测试	严格使用测试
主要用于返回数据	主要用于验证应用程序请求的精确用法
可以创建多个后端定义，与请求顺序无关	创建多个请求期望时需要注意请求顺序
后端定义匹配完一个请求后可以继续匹配下一个请求	当预期的请求出现后，请求期望会从期望列表中删除，无法重用

大多数情况下开发人员使用when，因为大多数测试只是测试应用程序如何处理$http返回的结果，而不是HTTP请求本身。

如果要验证HTTP请求本身，可以使用expect，它可以：

- 期望应用程序按指定顺序发出HTTP请求。
- 期望应用程序发出指定数量的HTTP请求。

假设需要测试应用程序是否按照指定顺序发出一系列的HTTP请求，其中必须发两次http://localhost/2请求，则相关测试代码如下：

```
it('should demonstrate using expect in sequence', function () {
   $httpBackend.expect('GET', 'http://localhost/1').respond(200);
   $httpBackend.expect('GET', 'http://localhost/2').respond(200);
   $httpBackend.expect('GET', 'http://localhost/2').respond(200);
   $httpBackend.expect('GET', 'http://localhost/3').respond(200);
   /* Code under test */
   $http.get('http://localhost/1');
   $http.get('http://localhost/2');
   $http.get('http://localhost/2');
   $http.get('http://localhost/3');
   /* End */
   expect($httpBackend.flush).not.toThrow();
});
```

在以上测试用例里使用$httpBackend.expect创建了4个请求期望，对应于4个HTTP请求。当$httpBackend.flush被调用时，这些请求期望按指定顺序对HTTP请求进行验证。一旦某个请求期望被满足，那么这个请求期望会从期望列表中清除。只要有一个请求期望没有得到满足或者有一个HTTP请求没有被验证，那么测试失败。例如以下的应用程序代码会导致测试失败，因为原本期望一次http://localhost/3请求，但是程序发出了两次请求：

```
/* Code under test */
$http.get('http://localhost/1');
$http.get('http://localhost/2');
$http.get('http://localhost/2');
$http.get('http://localhost/3');
$http.get('http://localhost/3');
/* End */
```

4. when和expect详解

when和expect函数有相同的参数（以下代码使用when演示示例），其中data、headers和keys是可选参数，格式如下：

```
when(method, url, [data], [headers], [keys]);
expect(method, url, [data], [headers], [keys]);
```

（1）method参数

method参数用来匹配应用程序使用的HTTP请求方法（如GET、POST、PUT、PATCH、DELETE、HEADER和JSONP）。

（2）url参数

url参数可以是字符串、正则表达式或者是一个回调函数function(url)，用来匹配HTTP请求的URL。

其中，回调函数是对URL进行的自定义匹配。如果URL符合匹配条件，那么回调函数返回true。以下示例代码期望HTTP请求的URL包含http://localhost/。

```
$httpBackend
   .when('GET', function (url) {
      return url.indexOf('http://localhost/') !== -1;
   })
   .respond(200, { name: 'foo' });
```

（3）data参数

可选参数data可以是字符串、正则表达式、对象或者是一个回调函数function(string)，用来匹配应用程序使用POST或PUT提交的数据。

以下示例代码期望应用程序提交的是一个标准的JavaScript对象：

```
$httpBackend
   .when('POST', 'http://localhost/api/products', {
   name: 'bob',
   description: 'bob_description'
})
.respond(201);
```

如果使用字符串或正则表达式作为data参数，这相当于将应用程序提交的对象序列化成JSON字符串然后再进行匹配。回调函数则是对提交的数据进行自定义匹配，如果数据符合条件，那么回调函数返回true。示例代码如下：

```
$httpBackend
   .when('POST', 'http://localhost/api/products'
      , function (data) {
      return angular.fromJson(data).name === 'bob';
   }).respond(201);
```

（4）headers参数

可选参数headers可以是一个对象或者是一个回调函数function(Object)，用来匹配应用程序发出请求的HTTP头。

以下示例代码期望应用程序发出的请求包含一个自定义的HTTP头字段myHeader，并且它的值是products：

```
$httpBackend
   .when('GET', 'http://localhost/foo/productinfo.json', undefined, {
      myHeader: 'products',
      Accept: 'application/json, text/plain, */*'
   })
   .respond(200, { name: 'foo' });
```

AngularJS默认[1]自动为所有请求添加HTTP头字段Accept: application/json、text/plain和

① AngularJS. $http[OL]. [2016]. https://docs.angularjs.org/api/ng/service/$http.

/。由于测试时对象需要完全匹配请求包含的所有字段，所以以上示例里headers对象也要包含Accept字段。

回调函数则是对HTTP头进行自定义匹配，如果HTTP头符合条件，那么回调函数返回true。示例代码如下：

```
$httpBackend
   .when('GET', 'http://localhost/foo/productinfo.json',
      undefined,
      function (headers) {
      return headers.myHeader === 'products';
   })
   .respond(200, { name: 'foo' });
```

（5）keys参数

keys是一个可选数组参数。如果使用url正则表达式来匹配URL，keys用来存储正则表达式的匹配项，供输出结果时（respond函数）使用。

When和expect函数针对各个HTTP方法都提供了对应的快捷方法，格式如下：

```
// when
whenGET(url, [headers], [keys]);
whenHEAD(url, [headers], [keys]);
whenDELETE(url, [headers], [keys]);
whenPOST(url, [data], [headers], [keys]);
whenPUT(url, [data], [headers], [keys]);
whenJSONP(url, [keys]);
// expect
expectGET(url, [headers], [keys]);
expectHEAD(url, [headers], [keys]);
expectDELETE(url, [headers], [keys]);
expectPOST(url, [data], [headers], [keys]);
expectPUT(url, [data], [headers], [keys]);
expectPATCH(url, [data], [headers], [keys]);
expectJSONP(url, [keys]);
```

5. 使用respond方法

when和expect函数都会返回一个对象，该对象具有一个respond方法。当应用程序的HTTP请求符合预先设置的响应条件时，可以用respond方法控制返回结果。

respond方法有以下两种格式：

```
respond([status,] data[, headers, statusText]);

respond(function(method, url, data, headers, params);
```

第一种格式直接通过参数的方式设定返回结果。各参数说明如下：

- status：HTTP状态码，例如200、201、404、500等。
- data：指定返回数据，例如JSON数据。
- headers：HTTP头。
- statusText：状态描述。

以下示例代码直接返回一个对象：

```
    var url = 'http://localhost/foo/productinfo.json';

    $httpBackend

        .when('GET', url)

        .respond(200, { name: 'foo' });
```

第二种格式使用回调函数，回调函数可以根据应用程序的请求内容动态创建返回结果，格式如下：

```
   function(method, url, data, headers, params)
```

传入回调函数的参数就是HTTP请求的内容，其中最后一个参数params是一个对象有以下两种情况：

- 默认情况下请求URL里的查询字符串（Query String）会被解析到params对象里。例如URL/list?foo=bar&baz=bla里的查询字符串会被解析成params对象{foo: 'bar', baz: 'bla'}。
- 如果when或expect使用正则表达式匹配URL，并且已经提供了keys参数，那么匹配项被保存在keys数组里，params对象里每个字段就对应着keys数组里每一项。在下面的示例代码中：

```
$httpBackend
   .when('GET', /localhost\/(.+)\/productinfo.json/,
      undefined,
      undefined,
      ['name']
      )
   .respond(function(method, url, data, headers, params) {
      return [200, {name: params.name}];
   });
```

如果HTTP请求的URL是http://localhost/foo/productinfo.json，经过以上示例代码中的正则表达式解析后，其匹配项是foo，保存到name里。所以respond回调函数的params对象是{name: 'foo' }。

回调函数的返回值是一个数组，包含status、data、headers和statusText。

6. 使用flush方法

在产品环境中，AngularJS程序对服务器端的HTTP请求都是异步的，但是在单元测试中，不太容易实现异步代码的测试。$httpBackend提供的flush方法允许测试立即响应等待的请求，这样就可以让异步请求同步化，从而能够在单元测试中同步测试HTTP请求。

单元测试里的HTTP请求被发送到$httpBackend里等待处理，这些HTTP请求按照请求顺序被响应。可以利用flush方法指定响应请求的数量，或者指示flush方法忽略一个或几个请求。其格式如下：

```
$httpBackend.flush([count], [skip]);
```

- Count：可选参数，用于指定flush函数响应请求的数量。如果不提供此参数，所有等待中的请求（从skip开始）都会被响应。
- Skip：可选参数，默认值为0，用于指定flush函数忽略的请求。例如skip是5，则flush函数会忽略前5个等待请求，从第6个开始响应。

flush方法被调用时，如果当时没有等待的HTTP请求，它会抛出一个异常。

7. 辅助方法

$httpBackend还提供了其他几种辅助方法以确保应用程序代码能正确处理HTTP请求。

- verifyNoOutstandingExpectation()：被调用时，如果任何expect设置的条件都没有匹配，它就会抛出异常。flush内部也会调用它。
- verifyNoOutstandingRequest()：被调用时，如果还有等待的请求没有被响应，它就会抛出异常。
- resetExpectations()：重置所有expect创建的请求期望（request expectation），但是保留when创建的后端定义（backend definition）。

通常在Jasmine的afterEach里调用verifyNoOutstandingExpectation和verifyNoOutstandingRequest方法，确保每个测试用例完成后所有expect设置的条件都被匹配，并且所有等待的请求都被响应。示例代码如下：

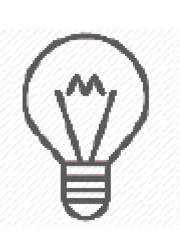
```
// make sure no expectations were missed in your tests.
afterEach(function () {
   $httpBackend.verifyNoOutstandingExpectation();
   $httpBackend.verifyNoOutstandingRequest();
});
```

6.3.4 测试Directive

Directive（指令）是AngularJS最重要也是最复杂的组件。AngularJS内建了丰富的Directive（例如ng-app和ng-controller等），也允许用户创建自定义的Directive。这里测试Directive指的是测试自定义Directive。

Directive在HTML里声明，作为DOM元素上的标记，通过AngularJS的HTML编译器使DOM元素拥有特定的行为，和用户进行交互。

一个Directive既有HTML模板，又有和用户进行交互的JavaScript代码，使得用户无法像调用函数一样使用Directive，因此对Directive进行单元测试变得非常棘手。接下来介绍针对Directive的各个部分分别进行的单元测试。（演示代码在C:\ngmock-demo\app\directivem目录下）



AngularJS的编程模式是声明式编程，大部分的DOM操作可以通过AngularJS提供的Directive来完成。如果应用程序需要进行DOM操作，建议创建自定义的Directive来封装DOM操作。

1. 测试DOM操作

首先准备一个简单的Directive，这个Directive的link函数会添加一段span到它声明所在的HTML元素中，具体代码如下：

```
angular.module('demoApp.directive')
    .directive('appendSpanDirective', function() {
        return {
            link: function(scope, elem) {
                elem.append('<span>It is appended from directive.</span>');
            }
        };
    });
```



AngularJS推荐在link函数里操作DOM。

和测试AngularJS其他组件一样，为了测试Directive，需要先获得Directive的实例。但是Directive无法通过注入直接获得，需要做一些特殊准备工作，代码如下：

```
    var $compile, $scope, directiveElem;
    beforeEach(function () {
        angular.mock.module('demoApp.directive');
        angular.mock.inject(function (_$compile_, _$rootScope_) {
            var element;
            $compile = _$compile_;
            $scope = _$rootScope_.$new();
            element = angular.element('<div append-span-directive></div>');
            directiveElem = $compile(element)($scope);
        });
    });
```

beforeEach函数做了如下的准备工作：

（1）加载Directive所在的模块demoApp.directive。

（2）注入$compile和$rootScope服务。这里注入$compile服务的原因是，在生产环境中应用程序被ng-app引导启动，AngularJS会遍历页面的DOM元素，识别所有的Directive并且调用$compile服务编译。但是在单元测试中，必须手动调用$compile对被测Directive进行编译。

（3）利用$rootScope创建一个新的Scope。

（4）因为Directive需要在HTML里声明，所以准备了测试用的div元素，并声明append-span-directive。

（5）调用$compile服务编译测试用的DOM元素和Directive。

准备工作完成后，测试用例验证appendSpanDirective在编译后是否生成span元素以及是否包含指定字符串。代码如下：

```
it('should have span element', function () {
    var spanElement = directiveElem.find('span');
    expect(spanElement).toBeDefined();
    expect(spanElement.text()).toEqual('It is appended from directive.');
});
```

2. 测试Watcher（监视器）

传统的前端JavaScript程序会进行大量的DOM操作，但是AngularJS的数据绑定功能使得开发人员从繁琐的DOM操作中解脱出来。例如使用以下这段代码，即可在页面上实时输出用户在文本输入框里输入的内容：

```
    <input type="text" ng-model="aModel"/>
    <div>{{aModel}}</div>
```

如果采用传统的方式完成相同功能，则可能需要监听文本输入框的各种事件，例如用户敲击的按键，并在每次事件发生后把文本框里的内容写入div元素中。

以上展示的AngularJS数据绑定功能的关键是利用了当前Scope的数据模型aModel。它通过内建的ng-model指令与输入框绑定，同时利用AngularJS表达式{{aModel}}与div元素绑定。AngularJS为aModel在当前Scope创建一个Watcher（监视器），一旦aModel数据有变化，Watcher会根据预先设定更新视图。

除了使用ng-model和AngularJS表达式，开发人员还可以调用$scope.$watch手动注册

Watcher要监视的内容。

以下示例里directive使用Watcher监视当前Scope里message内容的变化：

```
angular.module('demoApp.directive')
    .directive('appendMessageDirective', function() {
        return {
            link: function(scope, elem) {
                var spanElement = angular.element(
                    '<span>' + scope.message + '</span>');
                elem.append(spanElement);
                scope.$watch('message', function(newVal, oldVal){
                    spanElement.text(newVal);
                });
            }
        };
    });
```

测试这个Directive和前面的appendSpanDirective很类似，但是此处需要验证当Scope里message变化时，Directive里的内容也要随之改变。

```
it('should have span element', function () {
    $scope.message = 'It is appended from directive.';
    $scope.$apply();
    var spanElement = directiveElem.find('span');
    expect(spanElement).toBeDefined();
    expect(spanElement.text()).toEqual('It is appended from directive.');
});
```

注意，以上测试用例在更新message内容后调用$scope.$apply函数，这个函数和AngularJS内部一个被称为$digest循环的机制有关，如图6-9所示。

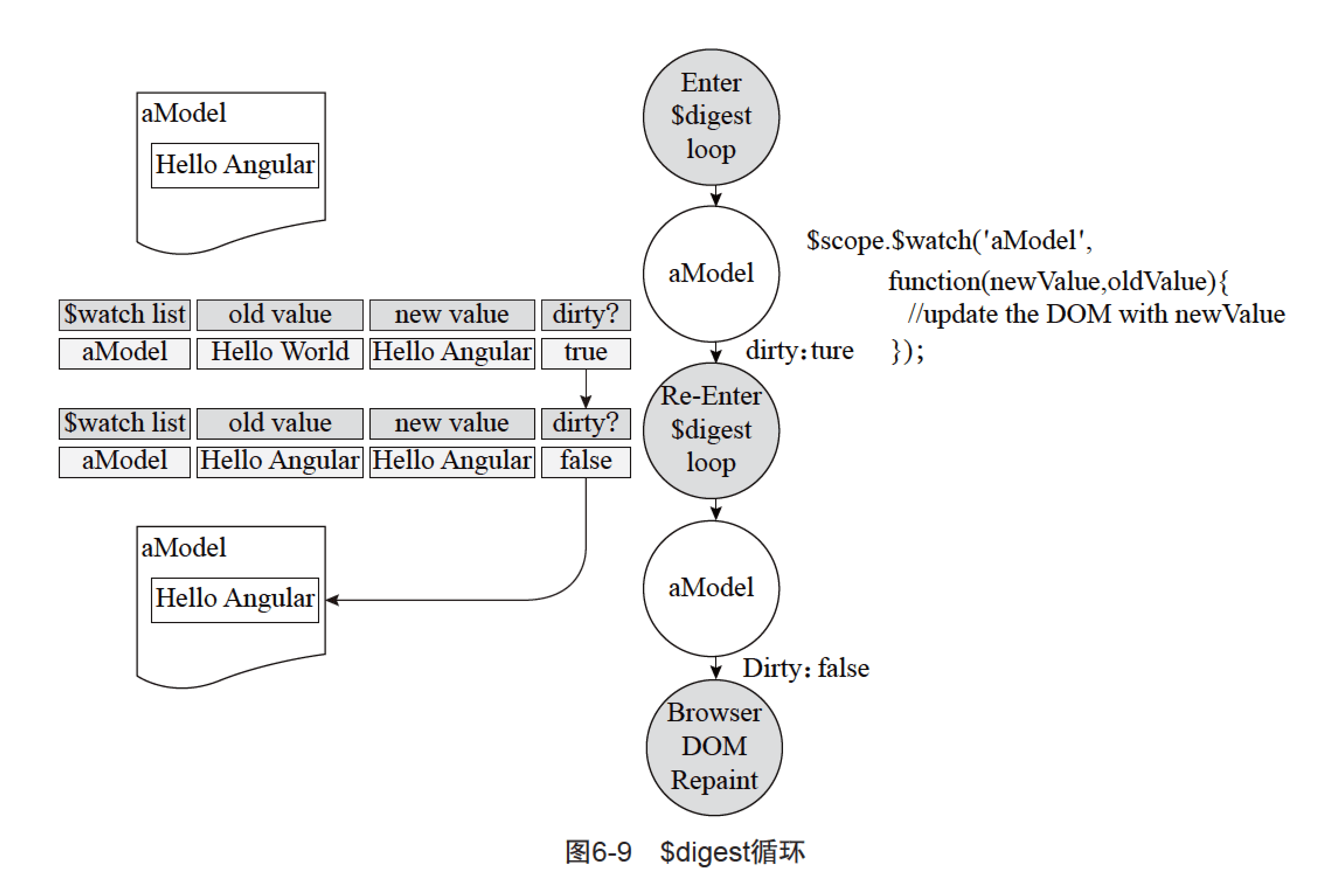


图6-9 $digest循环

$digest循环的目的是检查网页上的数据模型是否发生了变化并更新数据，在数据趋于稳定的情况下，统一渲染页面，这样可以避免为了响应某个数据变化而不停刷新页面，提高应用程序的性能。AngularJS内部是由$scope.$digest函数触发$digest循环。很多内建的允许改变数据模型的Directive（例如ng-model、ng-click）在用户更新数据后都会自动调用$scope.$digest函数触发循环。前面提到AngularJS使用Watcher监视数据模型的变化，一旦$digest循环开始，AngularJS就会依次启动各个Watcher。每个Watcher比较它所监视的Scope的数据模型和上次值是否相同。如果不同的话，就会执行Watcher的回调函数进行相应操作（例如示例代码将span元素的内容改成了新内容）。当所有Watcher遍历完后，AngularJS会重新遍历Watcher列表，直到所有被监视的数据模型没有再次被改动。

在上面的单元测试中更新了$scope.message的内容，但是它不会自动触发$digest循环，所以需要手动调用$scope.$apply函数进行触发，以确保在断言前Directive的内容得到更新。



通常不直接调用$scope.$digest函数，而是使用$scope.$apply。$scope.$apply内部会调用$rootScope.$digest，这样$digest循环会从$rootScope开始，启动它下面所有Scope里的Watcher。

3. 测试DOM事件

Directive通过增强DOM元素处理各种DOM事件，使用户交互变得简单有效。在AngularJS中这些用户交互代码也可以被测试。例如以下代码的incrementValueDirective里有一个按钮，一旦单击按钮，则value的值加1：

```
angular.module('demoApp.directive')
    .directive('incrementValueDirective', function () {
        return {
            template: '<button>Increment value!</button>',
            link: function (scope, elem) {
                elem.find('button').on('click', function () {
                    scope.value++;
                });
            }
        };
    });
```

测试这个Directive需要模拟单击设定的按钮，然后验证value的值，代码如下：

```
it('should increment value on click of button', function () {
    $scope.value = 5;
    var button = directiveElem.find('button');
    button.triggerHandler('click');
    $scope.$apply();
    expect($scope.value).toEqual(6);
});
```

AngularJS内建了一个轻型的jQuery库，称为jQuery lite或者jqLite[①]。以上测试用例调用jqLite的triggerHandler函数触发按钮单击事件，然后验证value的值。

但是jqLite只提供jQuery的部分功能。如果应用程序引用了jQuery，那么AngularJS就会使用jQuery取代jqLite。我们在Karam的配置文件里可以添加jQuery库文件，即可在单元

① AngularJS. API: angular.element[OL]. [2016]. https://docs.angularjs.org/api/ng/function/angular.element.

测试时使用更多jQuery的功能。

```
files: [
    'node_modules/jquery/dist/jquery.js',
    'node_modules/angular/angular.js',
    'node_modules/angular-mocks/angular-mocks.js',
    'app/**/*.js',
],
```

4. 测试使用模板的Directive

Directive使用模板有两种方式：内联和使用模板文件。已介绍的示例incrementValue Directive就使用了内联模板。因为内联模板和Directive在同一个文件内，所以测试这样的Directive不需要额外的步骤。

如果Directive使用模板文件，那么AngularJS需要发出一个异步的HTTP请求获取这个模板文件。例如：

```
angular.module('demoApp.directive')
    .directive('productInfoDirective', function () {
        return {
            templateUrl: 'app/directive/product-info.html'
        };
    });
```

在单元测试时，模板必须加载完才能验证测试结果，但是AngularJS并没有提供一个通知模板加载完毕的事件。那么如何才能测试使用模板文件的Directive呢？

在AngularJS里，当一个模板首次使用时，它会被加载到$templateCache[①]模板缓存中，这样将来应用程序使用这个模板时可以从缓存直接获取。应用程序也可以通过script标签或者调用$templateCache服务的方式直接把文件加载到缓存里。

为了测试使用模板文件的Directive，需要在测试前由Karma将模板文件预先加载到AngularJS的模板缓存，这样在测试时，Directive会直接从$templateCache里获取模板文

① AngularJS. API: $templateCache[OL]. [2016]. https://docs.angularjs.org/api/ng/service/$templateCache.

件，而不需要连接外部服务器。Karma提供了插件karma-ng-html2js-preprocessor[①]来满足这个需求。

使用npm命令安装该插件，代码如下：

```
C:\ngmock-demo>npm install karma-ng-html2js-preprocessor --save-dev
```

修改项目根目录下的c:\ngmock-demo\karma.conf.js文件。首先向其中增加一个字段preprocessors，告诉karma测试前使用预处理器ng-html2js：

```
preprocessors: {
      'app/directive/*.html': ['ng-html2js']
},
```

然后在files字段添加需要加载的html文件：

```
files: [
   'node_modules/jquery/dist/jquery.js',
   'node_modules/angular/angular.js',
   'node_modules/angular-mocks/angular-mocks.js',
   'app/**/*.js',
   'app/directive/*.html'
],
```

最后添加字段ngHtml2JsPreprocessor对插件进行设置：

```
ngHtml2JsPreprocessor: {
 moduleName: 'demoApp.template'
},
```

插件karma-ng-html2js-preprocessor的作用是将HTML模板文件转成JavaScript代码，生成一个AngularJS的模块，模块名可以通过ngHtml2JsPreprocessor字段指定，本例是demoApp.template。执行时，加载这个模块（demoApp.template），运行模块内的

① Karma. A Karma plugin. Compile AngularJS 1.x and 2.x templates to JavaScript on the fly[OL]. [2016]. https://github.com/karma-runner/karma-ng-html2js-preprocessor.

JavaScript代码将模板文件内容放入$templateCache中，这样AngularJS无需连接外部服务器就可以读取模板文件。

karma-ng-html2js-preprocessor会生成什么样的JavaScript代码呢？

示例使用的模板文件product-info.html内容如下：

```
<h3>Product Name: {{product.name}}<h3>
```

karma-ng-html2js-preprocessor会将模板文件转换成以下的JavaScript代码：

```
(function (module) {
   try {
      module = angular.module('demoApp.template');
   } catch (e) {
      module = angular.module('demoApp.template', []);
   }
   module.run(['$templateCache', function ($templateCache) {
      $templateCache.put('app/directive/product-info.html',
         '<h3>Product Name: {{product.name}}<h3>');
   }]);
})();
```

被缓存的模板文件都有一个键值，这个键值是模板文件从服务器处下载的路径：

```
      $templateCache.put('app/directive/product-info.html',
         '<h3>Product Name: {{product.name}}<h3>');
```

本示例里Karma下载模板文件的路径是app/directive/product-info.html，所以缓存里的键值也是app/directive/product-info.html。但是这可能和运行时候AngularJS获取模板的路径不同。

运行时AngularJS从Directive的templateUrl字段指定位置处下载文件，templateUrl路径是相对于引用Directive的HTML文件。本书示例假设引用productInfoDirective的HTML文件在项目根目录c:\ngmock-demo，所以templateUrl是app/directive/product-info.html，恰巧和Karma读取的文件路径一致。

如果引用productInfoDirective的HTML文件在c:\ngmock-demo\app目录，那么templateUrl就需要修改为directive/product-info.html，这种情况下单元测试会试图使用路径directive/product-info.html在缓存里寻找模板，但缓存里的键值却是app/directive/product-info.html。为了解决该问题，可以设置karma-ng-html2js-preprocessor，改变缓存的键值：

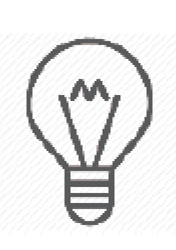
```
ngHtml2JsPreprocessor: {
     // strip this from the file path
     stripPrefix: 'app/',
     moduleName: 'demoApp.template'
   },
```

以上示例里将下载路径去掉app/后作为键值，这样键值即与templateUrl保持相同。

有了karma-ng-html2js-preprocessor的帮助，测试productInfoDirective就变得简单多了，代码如下：

```
/* product-info.directive.spec.js */
describe('productInfoDirective', function() {
   var $compile, $scope, directiveElem;
   beforeEach(function() {
      angular.mock.module('demoApp.template');
      angular.mock.module('demoApp.directive');
      angular.mock.inject(function(_$compile_, _$rootScope_) {
         var element;
         $compile = _$compile_;
         $scope = _$rootScope_.$new();
         element = angular.element('<div product-info-directive></div>');
         directiveElem = $compile(element)($scope);
         $scope.$apply();
      });
   });
   it('should applied template', function() {
      var h3Element = directiveElem.find('h3');
```

```
        $scope.product = { name: 'foo' };
        $scope.$apply();
        expect(h3Element.text()).toEqual('Product Name: foo');
    });
});
```



测试Directive需要注意以下两点：

- 需要调用angular.mock.module加载karma-ng-html2js-preprocessor生成的模块。该模块的JavaScript代码会把模板文件内容加入缓存。
- $compile编译完Directive后，需要调用$scope.$apply触发$digest循环。

5. 测试Directive的Scope

Directive的Scope分3种类型：

- 共享：Directive使用声明它的HTML元素的Scope。
- 继承：Directive创建一个Scope，新的Scope继承自声明Directive的HTML元素的Scope。
- 独立：创建一个本地的独立Scope。

测试Scope就是验证Scope对象的状态是否按预期改变。使用前两种Scope的Directive天然可以和外界交换数据，但是对于本地独立的Scope来说，需要有另外的机制。以下示例代码定义了一个使用独立Scope的Directive：

```
angular.module('demoApp.directive')
    .directive('isolatedScopeDirective', function () {
        return {
            scope: {
                config: '=',
                notify: '@',
                onChange: '&'
            }
        };
    });
```

这个独立Scope有3个字段，使用了不同的前缀标识符来引用声明Directive的HTML元素上的属性：

- config：“=”是双向数据绑定前缀标识符。此字段用于使Directive本地Scope的字段值和声明Directive的HTML元素的Scope保持同步。
- notify：“@”是单向数据绑定前缀标识符。此字段用于声明Directive的HTML元素可以通过notify属性传入数据。修改Directive本地Scope的notify字段值不会影响外面元素的行为。
- onChange：“&”是绑定函数方法的前缀标识符。在Directive内调用onChange时也会调用声明Directive的HTML元素上绑定的函数方法。

为了测试这个使用独立Scope的Directive，需要准备相应的测试数据并且编译声明Directive的元素。示例代码如下：

```
var $compile, $scope, directiveElem;
beforeEach(function () {
   angular.mock.module('demoApp.directive');
   angular.mock.inject(function (_$compile_, _$rootScope_) {
      var element;
      $compile = _$compile_;
      $scope = _$rootScope_.$new();
      $scope.config = {
         prop: 'value'
        };
        $scope.notify = true;
        $scope.onChange = jasmine.createSpy('onChange');
         element = angular.element('<isolated-scope-directive config="config" notify=
"notify" on-change="onChange()"></isolated-scope.directive>');
        directiveElem = $compile(element)($scope);
        $scope.$apply();
   });
});
```

测试用例如下：

```
it('config on isolated scope should be two-way bound', function () {
   var isolatedScope = directiveElem.isolateScope();
   isolatedScope.config.prop = "value2";
   expect($scope.config.prop).toEqual('value2');
});
it('notify on isolated scope should be one-way bound', function () {
   var isolatedScope = directiveElem.isolateScope();
   isolatedScope.notify = false;
   expect($scope.notify).toEqual(true);
});
it('onChange should be a function', function () {
   var isolatedScope = directiveElem.isolateScope();
   expect(typeof (isolatedScope.onChange)).toEqual('function');
});
it('should call onChange method of scope when invoked from isolated scope', function () {
   var isolatedScope = directiveElem.isolateScope();
   isolatedScope.onChange();
   expect($scope.onChange).toHaveBeenCalled();
});
```

以上示例通过调用angular.element.isolateScope函数获得Directive的本地独立Scope，通过修改独立Scope里字段的值来验证独立Scope的行为。

6.3.5 测试$timeout和$interval

AngularJS内建的$timeout和$interval组件封装了JavaScript的setTimeout和setInterval函数，所以测试使用内建$timeout和$interval的AngularJS应用时，一方面开发人员希望单元测试尽可能快速运行（“快进”时钟），另一方面也希望在执行断言时所有的异步回调函数已经被执行完毕。为了实现这个目的，ngMock提供了对应的测试专用$timeout和

$interval组件，可以在单元测试中替代内建的$timeout和$interval，协助完成测试。

1. $timeout

执行单元测试时，ngMock测试专用$timeout会替换AngularJS内建的$timeout，并且它额外提供了如表6-4所示的方法。

表6-4　ngMock的$timeout方法

方　法	描　述
flush([delay])	强制执行所有等待的$timeout回调函数。通过可选参数delay指定在强制执行前等待（“快进”）的毫秒数
verifyNoPendingTasks()	执行该方法时，如果还有等待的回调函数没有被执行，那么该函数抛出异常

以下示例注入了一个ngMock的$timeout实例，然后调用它的flush方法立即执行所有$timeout的回调函数，确保在执行断言expect函数时回调函数已经被执行。

```
/* timeout.spec.js */
describe('Controller', function () {
   var $controller;
   var $timeout;
   beforeEach(function () {
      angular.mock.inject(function (_$controller_, _$timeout_) {
         $controller = _$controller_;
         $timeout = _$timeout_;
      });
   });
   it('should set result to 5 with timeout', function () {
      var timeoutController;
      timeoutController = $controller(function ($timeout) {
         var vm = this;
         $timeout(function () {
            vm.result = 5;
         }, 3000);
      }, {});
      $timeout.flush();
      // this will throw an exception if there are any pending timeouts.
```

```
        $timeout.verifyNoPendingTasks();
        expect(timeoutController.result).toBe(5);
    });
});
```

2. $interval

执行单元测试时，ngMock的测试专用$interval会替换AngularJS内建的$interval。ngMock的$interval本身可以作为一个函数使用，同时它额外提供如表6-5所示的方法协助测试。

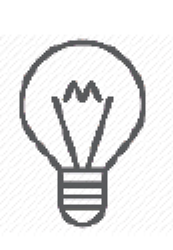
表6-5　ngMock的$interval方法

方　　法	描　　述
cancel(promise)	取消和promise关联的任务。通常使用$interval函数建立一个任务，该函数返回一个promise。如果要取消这个任务， 那么可以将这个promise传给cancel方法
flush([milliseconds])	强制执行所有等待的$interval任务。通过可选参数可指定强制执行前等待（“快进”）的毫秒数

以下示例代码定义了CounterController，执行start函数启动$interval任务，计数器每秒加1，10秒后停止。CounterController也提供了cancel函数用来取消$interval任务（C:\ngmock-demo\app\interval\counter.controller.js）。

```
/* counter.controller.js */
(function () {
    angular.module('demoApp.interval', []);
    angular.module('demoApp.interval')
        .controller('CounterController', CounterController);
    CounterController.$inject = ['$scope', '$interval'];
    function CounterController($scope, $interval) {
        var vm = this;
        var counterInstance;
        vm.counter = 0;
        vm.counterFunction = function () {
            vm.counter += 1;
        };
        vm.start = function () {
```

```
            counterInstance = $interval(vm.counterFunction, 1000, 10);
        };
        vm.cancel = function () {
            if (angular.isDefined(counterInstance)) {
                $interval.cancel(counterInstance);
                counterInstance = undefined;
            }
        };
        // listen on DOM destroy (removal) event, and cancel the next UI update
        // to prevent updating time after the DOM element was removed.
        $scope.$on('$destroy', function () {
            vm.cancel();
        });
    }
})();
```



对于像$interval这样的AngularJS原生组件，大多数情况下只需要测试应用程序是否使用正确的参数来调用它们，而不会测试$interval本身的行为，因为这些组件是由AngularJS提供的。

以下示例调用jasmine.createSpy创建了一个新的spy函数intervalSpy以跟踪$interval函数。intervalSpy并不会调用实际的$interval，只是用来验证应用程序是否使用正确的参数来调用$interval。

```
    var $controller, $interval, $scope;
    beforeEach(function () {
        angular.mock.module('demoApp.interval');
        angular.mock.inject(function (_$controller_,
            _$interval_,
            _$rootScope_) {
            $controller = _$controller_;
            $interval = _$interval_;
```

```
        $scope = _$rootScope_.$new();
    });
});
it('should register the intervals', function () {
    var intervalSpy = jasmine.createSpy('$interval', $interval);
    var counterController = $controller('CounterController',
        { $scope: $scope, $interval: intervalSpy });
    counterController.start();
    /* Assertions */
    expect(intervalSpy).toHaveBeenCalled();
    expect(intervalSpy).toHaveBeenCalledWith(
        counterController.counterFunction,
        1000,
        10);
});
```

有时候开发人员也会测试$interval任务的执行情况。例如下面的示例代码：

```
it('should add 1 to counter every second', function () {
    // Note that we've added .and.callThough();
    var intervalSpy = jasmine.createSpy('$interval', $interval)
        .and.callThrough();
    // we need to register a spy for $interval's cancel function.
    spyOn(intervalSpy, 'cancel');
    var counterController = $controller('CounterController',
        { $scope: $scope, $interval: intervalSpy });
    counterController.start();
    // advance in time by 1 second from call to start()
    $interval.flush(1100);
    expect(counterController.counter).toBe(1);
    // advance in time by 4 second from call to start()
```

```
    $interval.flush(3000);
    expect(counterController.counter).toBe(4);
    counterController.cancel();
    expect(intervalSpy.cancel.calls.count()).toBe(1);
});
```

在创建spy函数时链式调用了.and.callThough()，这样intervalSpy不仅跟踪了$interval的使用情况，而且会调用实际的$interval；两次调用flush方法快进“时钟”，然后验证计数器的值。

6.3.6 测试Promise

Promise是JavaScript异步编程的一种设计模式，也是AngularJS使用的模式。AngularJS中所有的Ajax请求默认都返回一个Promise对象。测试HTTP交互示例中创建的basicHttpFactory，其getProductName方法返回的就是Promise对象（也是$http.get返回值）。示例代码如下：

```
function basicHttpFactory($http) {
    return {
        getProductName: getProductName
    };

    function getProductName(url) {
        return $http.get(url)
            .then(function (result) {
                return result.data.name;
            });
    }
}
```

这里，创建一个使用basicHttpFactory的Controller（C:\ngmock-demo\app\promise\product.controller.js），代码如下：

```
/* product.controller.js */
(function() {
   angular.module('demoApp.promise', ['demoApp.http']);
   angular.module('demoApp.promise')
      .controller('ProductController', ProductController);
   ProductController.$inject = ['basicHttpFactory'];
   function ProductController(basicHttpFactory) {
      var vm= this;
      basicHttpFactory.getProductName(
         'http://localhost/foo/productinfo.json')
         .then(function(data) {
            vm.name = data;
         }, function () {
            vm.error = 'There has been an error!';
         });
   }
})();
```

当数据获取成功时，将数据赋值于name，如果失败，则定义一个错误信息error。为了测试这个使用Promise的Controller，需要做一些准备工作，代码如下：

```
   var $scope, deferred, productController;
   beforeEach(function () {
      angular.mock.module('demoApp.promise');
      angular.mock.inject(function (_$controller_,
         _$q_, basicHttpFactory, _$rootScope_) {
         // We use the $q service to create a mock instance of defer
         deferred = _$q_.defer();
         $scope = _$rootScope_.$new();
         // Use a Jasmine Spy to return the deferred promise
```

```
        spyOn(basicHttpFactory, 'getProductName')
           .and.returnValue(deferred.promise);
        productController = _$controller_('ProductController', {
           basicHttpFactory: basicHttpFactory
        });
     });
  });
```

以上代码在beforeEach中做了如下准备工作：

（1）注入AngularJS的$q组件。$q是AngularJS中自己封装的一种Promise实现。这里利用它创建一个受控的deferred对象。

（2）在basicHttpFactory.getProductName方法上注册一个Jasmine spy函数。当应用程序调用该方法时，返回一个受控的Promise对象。

（3）创建Controller实例。

测试用例通过调用受控的deferred对象的resolve和reject方法来改变Promise的状态，验证Controller的执行结果。

```
it('should resolve promise', function () {
   // Setup the data we wish to return for the .then in the controller
   deferred.resolve('foo');
   // We have to call apply for this to work
   $scope.$apply();
   expect(productController.name).not.toBe(undefined);
   expect(productController.name).toBe('foo');
   expect(productController.error).toBe(undefined);
});
it('should reject promise', function () {
   deferred.reject();
   // We have to call apply for this to work
   $scope.$apply();
   expect(productController.name).toBe(undefined);
```

```
    expect(productController.error).toBe('There has been an error!');
});
```

6.3.7 测试$log

AngularJS的$log组件用来输出日志信息。AngularJS推荐使用$log取代console.log，示例代码如下：

```
var $log;
beforeEach(function () {
   angular.mock.inject(function (_$log_, _$controller_) {
      $log = _$log_;
      _$controller_(function () {
         $log.log('standard log');
         $log.info('info log');
         $log.error('error log');
         $log.warn('warn log');
         $log.debug('some debug information');
      });
   });
})
```

ngMock提供了对应的测试专用$log，用于在单元测试时替换内建的$log组件。ngMock的$log提供两个额外的方法协助测试，如表6-6所示。

表6-6 ngMock的$log方法

方　　法	描　　述
reset()	清除所有日志信息
assertEmpty()	检查有没有日志信息被记录。如果有日志信息，该方法抛出异常

测试用例如下：

```
it('should write to log', function () {
   expect($log.log.logs[0]).toEqual(['standard log']);
```

```
        expect($log.info.logs[0]).toEqual(['info log']);
        expect($log.error.logs[0]).toEqual(['error log']);
        expect($log.warn.logs[0]).toEqual(['warn log']);
        expect($log.debug.logs[0]).toEqual(['some debug information']);
    });
    it('should not call log (using reset)', function () {
        // this clears the logs
        $log.reset();
        expect($log.assertEmpty).not.toThrow();
    });
```

6.3.8　测试$exceptionHandler

如果在AngularJS表达式里有未被捕获的异常，那么这个异常会被$exceptionHandler处理。$exceptionHandler默认的实现是调用$log.error把异常信息输出到浏览器的终端窗口。

ngMock提供了对应的测试专用$exceptionHandler，它不使用$log，而是将异常信息保存在一个数组里（errors字段）。可以通过$exceptionHandlerProvider改变这个默认行为，让$exceptionHandler重新抛出异常。示例代码如下：

```
beforeEach(module(function($exceptionHandlerProvider) {
        $exceptionHandlerProvider.mode('log');
        /* OR */
        $exceptionHandlerProvider.mode('rethrow');
}));
```

以下是测试代码（C:\ngmock-demo\app\exception\exception.spec.js）：

```
/* exception.spec.js */
describe('$exceptionHandler', function () {
    var $log, $exceptionHandler, $timeout;
    beforeEach(function () {
        angular.mock.module(function ($exceptionHandlerProvider) {
```

```
        $exceptionHandlerProvider.mode('log');
      });
      angular.mock.inject(function (
        _$log_,
        _$exceptionHandler_, _$timeout_) {
        $log = _$log_;
        $exceptionHandler = _$exceptionHandler_;
        $timeout = _$timeout_;
      });
    });
    it('should demonstrate exception handling with log mode', function () {
      $timeout(function () { $log.log(1); });
      $timeout(function () { $log.log(2); throw 'banana peel'; });
      $timeout(function () { $log.log(3); });
      expect($exceptionHandler.errors).toEqual([]);
      expect($log.assertEmpty());
      $timeout.flush();
      expect($exceptionHandler.errors).toEqual(['banana peel']);
      expect($log.log.logs).toEqual([[1], [2], [3]]);
    });
  });
```

以上示例主要做了下列工作：

（1）使用$exceptionHandlerProvider设置log模式，这样异常信息会被保存在$exceptionHandler实例的errors字段中。

（2）在$timeout的回调函数里抛出异常，这是为了演示发生在异步代码里的异常。

（3）访问$exceptionHandler.errors验证异常信息。

第7章 代码覆盖率

代码覆盖率（Code Coverage）是软件测试中的一种审计标准，描述程序中源代码被测试的比例和程度。它是衡量测试工作进展情况的重要指标，通常被用来发现没有被测试覆盖的代码，但是它本身不能完全用来衡量代码质量。

本章将介绍：

- 代码覆盖率的衡量标准
- 代码覆盖率的意义
- JavaScript代码覆盖率工具Istanbul
- 使用Karma生成覆盖率报告

7.1 代码覆盖率的衡量标准

代码覆盖率的衡量标准有很多种，这里介绍最常用的几种。

7.1.1 函数覆盖率（Function Coverage）

函数覆盖率，顾名思义，就是衡量应用程序里每个函数是否被测试代码调用过。

以下面的JavaScript函数为例，这个函数原本只是将参数x和y相加，但是开发人员不小心写成x+y/y，所以这是个有缺陷的函数。

```
var inc = function (x, y) {
   if (y == undefined) y = 1;
```

```
    return x + y/y;
};
```

如果想要达到100%函数覆盖率，只需一个测试，例如调用inc(1)就可以了。如果函数无法被测试代码所覆盖，那就应该考虑这个函数是否真的需要了。

7.1.2 语句覆盖率（Statement Coverage）

语句覆盖率，有人也把它称为行覆盖率（Line Coverage），这是最常见的一种覆盖方式，就是衡量被测代码中每条可执行语句是否被测试所覆盖。

例如测试中调用：

```
inc(1);
```

inc函数中每一条语句（包括y=1;）都执行一次，它的语句覆盖率就是100%。

如果开发人员仅仅满足于100%语句覆盖率，不再继续编写其他测试用例对函数进行测试，那么函数里的缺陷（return x+y/y;）就无法被发现。

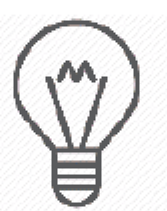
在已经满足100%语句覆盖率的情况下，继续调用：

```
inc(1,2);
```

或者：

```
inc(1, 0);
```

则函数里的问题就被暴露出来，所以语句覆盖常被人指责为“最弱的覆盖”，它只负责覆盖代码中的执行语句，却不考虑各种分支的组合。即使覆盖率达到100%，也很难换来好的测试效果。因此不能仅仅使用语句覆盖率来衡量软件测试的质量。



语句覆盖率和行覆盖率其实是有区别的。例如以下这行代码：

```
x=1;y=1;
```

虽然只有一行（line）代码，但是有两条语句（statement）。最后统计的覆盖率两者可能会不同。

7.1.3　分支覆盖率（Branch Coverage）

分支覆盖率，也称为判定路径 (decision-to-decision path，或DD-path) 覆盖率。它衡量程序中每一个判定的分支是否都被测试到。

例如测试中调用：

```
inc(1);
inc(1,2);
```

则测试代码满足分支覆盖率100%的条件。前者会使if的条件成立，后者使if的逻辑表达式条件（y==undefined）不成立。

7.1.4　条件覆盖率（Condition Coverage）

条件覆盖率指的是分支中的每个条件（与，或，非逻辑运算中的每一个条件判断）是否被测试所覆盖，每一个逻辑表达式中的每一个条件（无法再分解的逻辑表达式）是否都有执行结果为true和false的情形。

100%条件覆盖率并不意味着100%分支覆盖率。为了说明分支覆盖率和条件覆盖率的区别，考虑以下代码：

```
if(a && b) {
   …
}
```

使用以下测试条件可以得到100%条件覆盖率：

- a=true; b=false;
- a=false; b=true;

这两个条件使得每个条件表达式（a和b）都有true和false的结果，但是它们都不会使if的逻辑表达式（a && b）成立，所以分支覆盖率不能达到100%。

7.2　代码覆盖率的意义

经常有人问这样的问题：“做单元测试时代码覆盖率应该达到多少？”。Martin

Fowler曾经写过一篇博客[①]来讨论这个问题，他指出：把测试覆盖（代码覆盖）作为质量目标没有任何意义，我们应该把它作为发现未被测试覆盖的代码的一种手段（参见图7-1）。

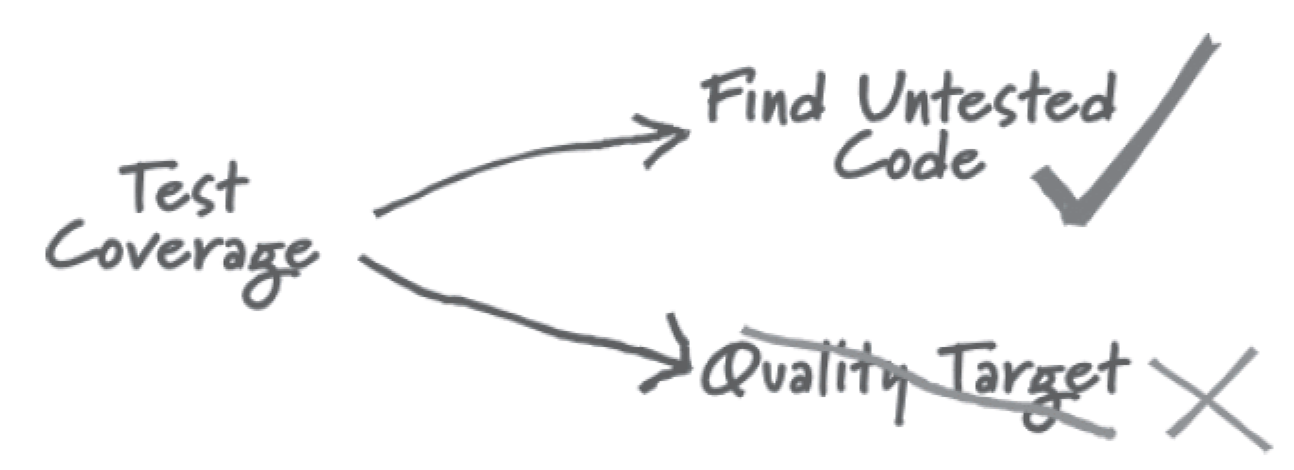


图7-1　Martin Fowler对代码覆盖的观点

这里阐述一下作者对代码覆盖率的看法：

（1）代码覆盖率是用来发现没有被测试覆盖的代码，不能完全用来衡量代码质量。

（2）高百分比的代码覆盖率不能代表高质量的测试，但是低百分比的代码覆盖率一定意味着测试不完整。

（3）通过分析未被测试覆盖的代码，可以验证前期测试设计是否充分。如果这些代码是测试设计的盲点，可以补充测试用例设计。

（4）不能盲目追求代码覆盖率。达到代码覆盖率目标是良好测试的结果，而不是目标本身。

7.3　JavaScript代码覆盖率工具Istanbul

Istanbul是目前比较流行的JavaScript测试覆盖率工具。它的名字取自土耳其的伊斯坦布尔，因为那里的地毯世界闻名，而地毯是用来覆盖的[②]。

Istanbul可以无缝地对JavaScript项目进行覆盖率统计。开发人员编写单元测试用例时，不需要为支持覆盖率统计编写额外的代码，测试用例也无需修改就可以直接使用Istanbul来统计覆盖率情况，而且Istanbul会生成一份漂亮的覆盖率报告，准确地标记出哪些代码没有被覆盖到。

---

① Martin Fowler. TestCoverage[OL]. 2012. http://martinfowler.com/bliki/TestCoverage.html.

② Krishnan Anantheswaran. Why the funky name[OL]. [2016]. https://github.com/gotwarlost/istanbul#why-the-funky-name.

7.3.1　安装Istanbul

Istanbul是一个npm软件包，可以使用如下npm命令进行全局安装：

```
npm install -g istanbul
```

运行istanbul help命令可列出istanbul命令的帮助信息：

```
C:\temp>istanbul help
Usage: istanbul help config | <command>
`config` provides help with istanbul configuration
Available commands are:
...
```

7.3.2　覆盖率测试

将以下代码保存为c:\temp\test.js。

```
/* test.js */
var inc = function (x, y) {
   if (y == undefined) y = 1;
   return x + y / y;
};
inc(1);
```

运行命令istanbul cover，得到代码覆盖率：

```
C:\temp>istanbul cover test.js
=================================================================
Writing coverage object [C:\temp\coverage\coverage.json]
Writing coverage reports at [C:\temp\coverage]
=================================================================
=============================== Coverage summary ================
```

```
Statements   : 100% ( 5/5 )

Branches     : 50% ( 1/2 )

Functions    : 100% ( 1/1 )

Lines        : 100% ( 4/4 )

========================================================================
```

Istanbul会统计4个覆盖率：语句覆盖率、分支覆盖率、函数覆盖率和行覆盖率。以上示例里有两个分支，但是inc(1)只执行了一个，所以分支覆盖率是50%，其他都是100%。

istanbul cover命令同时在当前目录下生成一个coverage子目录，其中的coverage.json[①]包含了覆盖率的原始数据。使用浏览器打开coverage\lcov-report\index.html文件，可以看到整个目录的覆盖率报告，如图7-2所示。

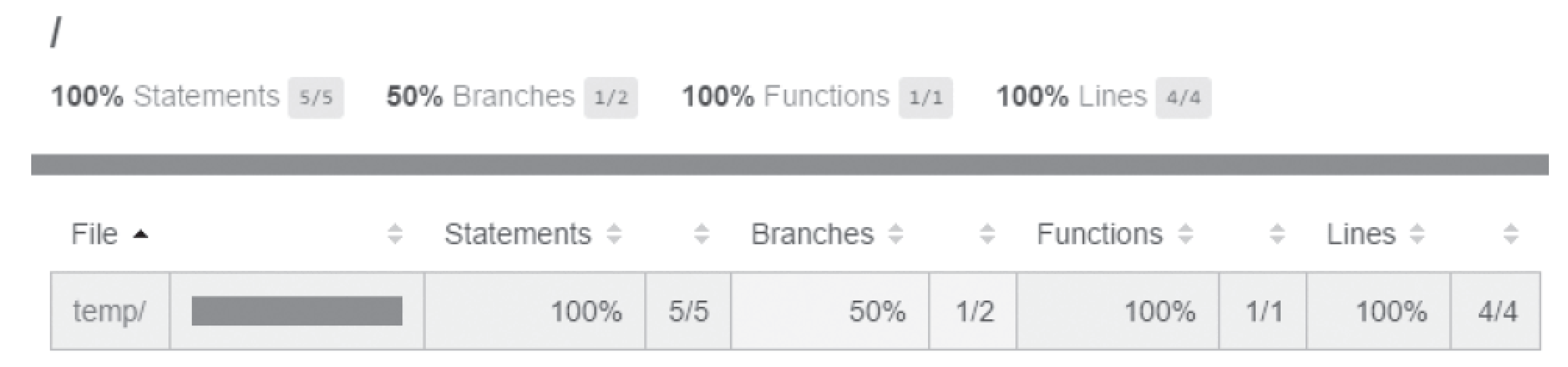


图7-2　Istanbul覆盖率报告（目录）

单击图7-2所示目录名temp，可以看到该目录下所有文件的覆盖率情况，如图7-3所示。

all files temp/

100% Statements 5/5　50% Branches 1/2　100% Functions 1/1　100% Lines 4/4

File ▲		Statements		Branches		Functions		Lines	
test.js		100%	5/5	50%	1/2	100%	1/1	100%	4/4

图7-3　Istanbul覆盖率报告（所有文件）

单击图7-3所示的test.js项，可以看到该文件详细的覆盖信息，如图7-4所示。其中标记E说明if语句的else分支没有被执行。

① Krishnan Anantheswaran. Format of coverage.json[OL]. [2016]. https://github.com/gotwarlost/istanbul/blob/master/coverage.json.md.

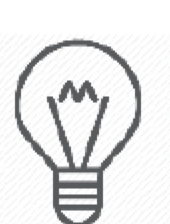
all files / temp/ test.js

100% Statements 5/5　50% Branches 1/2　100% Functions 1/1　100% Lines 4/4

```
   /* test.js */
1× var inc = function (x, y) {
1×     E if (y == undefined) y = 1;
1×     return x + y / y;
   };

1× inc(1);
```

图7-4　Istanbul覆盖率报告（单个文件）



在文件覆盖率报告里：

- 标识E说明在if语句里，if分支已经执行过但是else分支没有执行。
- 标识I和标识E相反。if分支没有执行过。
- 最左边那列Nx标记（例如1x）说明这行语句被执行过的次数。
- 没有被执行过的行或代码会用红色加亮。

7.3.3　覆盖率阈值

istanbul check-coverage命令用来设置阈值，同时将coverage.json里的结果和阈值进行比较，检查当前代码是否达标。

```
C:\temp>istanbul check-coverage --branches 90
ERROR: Coverage for branches (50%) does not meet global threshold (90%)
```

以上命令设置了分支覆盖率阈值90%。显然前面的覆盖率统计没有通过，因为它的分支覆盖率是50%。

除了百分比阈值，还可以使用负数设置绝对值阈值，例如：

```
istanbul check-coverage --statements -1
```

以上命令表示只允许有一个未被覆盖的指令。

7.3.4　忽略代码

Istanbul提供注释语法，允许部分代码不计入覆盖率。

- 忽略if分支，代码如下：

```
/* istanbul ignore if  */
if (hardToReproduceError)) {
   return callback(hardToReproduceError);
}
```

- 忽略else分支，代码如下：

```
/* istanbul ignore else  */
if (foo.hasOwnProperty('bar')) {
   // do something
}
```

- 忽略注释后面的内容，代码如下：

```
var object = parameter || /* istanbul ignore next */ {};
```

以上代码是为object指定默认值（一个空对象）。如果不想为object是空对象的情况写测试，可以用注释/* istanbul ignore next */，不将这种情况计入覆盖率。

7.3.5 Istanbul工作原理

Istanbul的工作原理如图7-5所示。

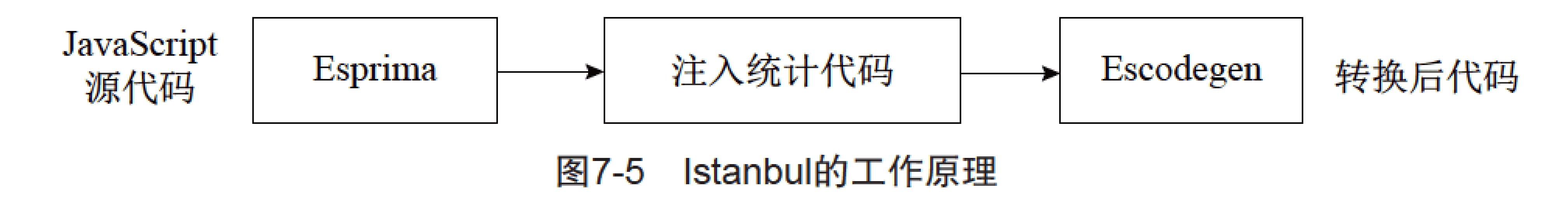


图7-5　Istanbul的工作原理

（1）使用开源工具Esprima[①]对JavaScript源代码进行语法分析，生成语法树。

（2）在语法树相应的位置注入统计代码。

（3）使用开源工具Escodegen[②]根据注入统计代码后的语法树生成对应的JavaScript代码，即转换之后的代码。

① Esprima. Esprima[OL]. [2016]. http://esprima.org/.
② Escodegen. ECMAScript code generator[OL]. [2016]. https://github.com/estools/escodegen.

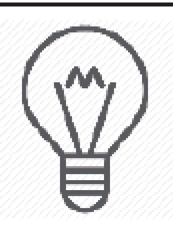
（4）执行转换后的代码。被注入的统计代码也会被执行，生成统计信息。

（5）根据覆盖率信息生成特定格式的报告。

7.4　使用Karma生成覆盖率报告

可以使用Istanbul命令生成覆盖率报告，也可以将这个功能集成到Karma中。如果想要Karma生成覆盖率报告，必须安装插件karma-coverage[1]，该插件底层使用Istanbul生成覆盖率报告。安装该插件的命令如下：

```
npm install karma-coverage --save-dev
```



如果使用karma-coverage，不需要全局安装Istanbul。

这里还是使用第6章测试AngularJS的项目。在c:\ngmock-demo目录下安装karma-coverage，然后修改karma.config.js文件。

（1）在proprocessors字段中添加coverage，代码如下：

```
preprocessors: {
   'app/directive/*.html': ['ng-html2js'],
   'app/**/!(*.spec).js':['coverage']
},
```

'app/**/!(*.spec).js'意思是除了*.spec.js以外的所有js文件。排除spec.js文件是因为这里只想统计应用程序代码的覆盖率，而不是测试用例代码的覆盖率。

（2）在reporters字段中添加 'coverage'。

```
reporters: ['progress','coverage'],
```

① karma-coverage. A Karma plugin Generate code coverage[OL]. [2016]. https://github.com/karma-runner/karma-coverage.

（3）建立一个新字段coverageReporter。这个字段设置报告的格式是HTML，输出路径是./report_output/coverage，代码如下：

```
coverageReporter:{
   type: 'html',
   dir: './report_output/coverage'
},
```

在命令控制台运行karma start，然后使用浏览器打开c:\ngmock-demo\report_output\coverage\<浏览器>\index.html，就能得到AngularJS项目在某个浏览器下运行的代码覆盖率报告，如图7-6所示。

/

100% Statements 86/86 100% Branches 2/2 100% Functions 40/40 100% Lines 86/86

File		Statements		Branches		Functions		Lines	
basic/		100%	10/10	100%	0/0	100%	5/5	100%	10/10
controller/		100%	7/7	100%	0/0	100%	2/2	100%	7/7
directive/		100%	24/24	100%	0/0	100%	16/16	100%	24/24
http/		100%	9/9	100%	0/0	100%	4/4	100%	9/9
interval/		100%	18/18	100%	2/2	100%	6/6	100%	18/18
module/		100%	3/3	100%	0/0	100%	1/1	100%	3/3
promise/		100%	9/9	100%	0/0	100%	4/4	100%	9/9
scope/		100%	6/6	100%	0/0	100%	2/2	100%	6/6

图7-6 Karma生成的代码覆盖率报告

如果在使用karma-coverage的情况下想要调试JavaScript代码，就会注意到源代码已经变得面目全非，如图7-7所示，这是因为此时的JavaScript代码是注入了统计代码并且经过转换的源代码。如果想要调试JavaScript源代码，建议在karma.conf.js里暂时注释掉karma-coverage的相关设置。

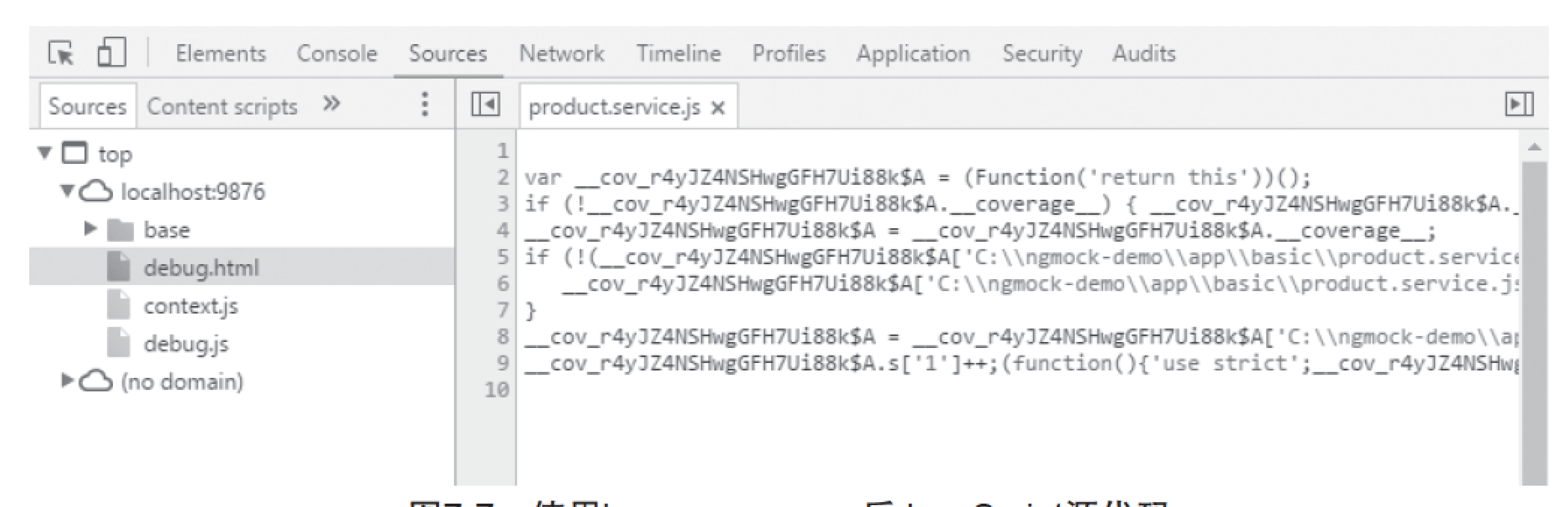


图7-7 使用karma-coverage后JavaScript源代码

自动化测试篇

第8章 走进自动化测试

长期以来，在Web应用的端到端测试中，手工测试一直占据绝大多数比例，原因是其简单易行，门槛较低。但在现代快速迭代的开发模式下，手工测试已经远远无法满足需求。大量的重复性工作，低下的测试效率导致软件质量无法保证。

特别是Web前端的测试环境复杂，兼容性要求高。很难想象有哪个公司能够持续投入巨大的人力成本，在手工测试领域全面覆盖Windows、Mac OS和Linux，以及不同的浏览器包括IE、Edge、Chrome和Firefox等。

本章将带领读者走进Web应用的自动化测试领域，了解端到端测试自动化的相关技术。

本章将介绍：

- 自动化测试的优势
- 自动化测试实施流程
- 自动化测试转型的适应性
- 测试工具的选择

8.1 自动化测试的优势

自动化的端到端测试是把传统的手工测试转化为机器执行的一种自动化过程，但并不会停止在端到端测试上的人力投入，恰恰相反，测试人员不仅不可或缺，而且将发展成为设计者，定义机器行为，让机器驱动测试。由于具体测试的执行者变为了机器，因此它可以持续地运行测试代码，节省了测试人员的重复劳动，为开发带来诸多手工测试无法企及的好处。

1. 充分利用硬件资源

理想的自动化测试能够按计划完全自动地运行，将烦琐的任务自动化。在开发人员和测试人员不可能实现24小时轮流工作的情况下，自动化测试可以一刻不停地运行。这样能够充分地利用硬件资源，避免开发和测试人员间的无效等待。

2. 充分调动测试人员的积极性

自动化测试使测试人员从烦琐的重复性工作中解脱出来，可以投入更多精力到自动化测试框架和用例的编写中。这将充分调动测试人员的工作热情，进一步提高测试的效率和可靠性。

3. 有效实施数据驱动的测试

自动化测试带来了测试效率的极大提升，使数据驱动的测试有了实施条件，让同样的测试用例使用不同的测试数据作为输入，提高了测试用例的广度。

4. 高效的回归测试

采用敏捷或迭代式开发意味着频繁的发布，必须用高频的回归测试来保证功能的完整和一致性。由于回归测试的动作和用例是已经设计好的，测试期望的结果完全可以预料的，因此回归测试自动化效果会非常显著，这不仅可以缩短测试时间，同时也能够让测试人员将更多的精力投入到新功能的开发测试中。

5. 一致性

因为每次测试运行的脚本是相同的，所以自动化测试保证了测试环境、测试路径的一致性，很容易发现被测软件的任何改变，而手工测试这是很难做到的。

6. 测试脚本具有可复用性

自动化测试通常采用脚本技术，编写脚本实际上是一个开发过程。一个软件尽管有多个功能点，但这些功能点在执行路径上会有重叠、复用的情况。所以只需要对相关测试脚本做少量修改，即可在不同的测试过程中重复使用。

8.2　自动化测试实施流程

自动化测试系统针对不同的实际项目，可以有不同的实施方法，但一般都由以下步骤组成。

1. 自动化测试适应性分析

虽然自动化测试有很多手工测试无法比拟的优点，但100%的自动化测试只是一个理想目标，一味追求自动化可能导致企业成本上升，效果甚至会低于手工测试。所以，在实施自动化测试之前需要进行适应性分析，明确当前的项目是否适合进行自动化测试。

2. 自动化测试需求分析

当项目满足了自动化测试的要求，并决定将在该项目中使用自动化测试技术后，即可开始进行自动化测试需求分析。此步骤需要确定自动化测试的范围以及相应的测试用例和测试数据，并形成详细的文档，以便于后续自动化测试框架的建立。

3. 自动化测试框架的搭建

搭建自动化测试框架就如进行软件架构一般，它定义了该测试框架的基础架构，构建环境以及实施工具。同时，测试框架也包括对公用环境的包装，即把各测试用例会用到的相同的要素独立包装，在各个测试用例中灵活调用，这样也能增强脚本的可维护性。

4. 设计与编写测试脚本

这一步骤是，基于已搭建好的自动化测试框架，针对每个测试用例编写测试脚本。

5. 测试的持续集成

持续集成是一个频繁持续在团队内进行业务集成、自我反馈的软件开发实践。在测试领域里，持续集成负责把下拉代码、编译、驱动自动化测试脚本和生成测试报告等步骤用一体化的方案进行构建。在配置持续集成的过程中，需要综合考虑持续集成工具的选择、与版本控制库的交互以及报告格式等。

在一个完善成熟的自动化测试解决方案里，以上5个步骤缺一不可，而且每个步骤都需要专业的开发测试人员参与其中，进行定制开发，这也正是测试人员发挥聪明才智的地方。简而言之，自动化测试将测试人员从重复低效的手工测试中解脱出来，角色转变为测试的设计者，指挥机器完成测试工作。

8.3 自动化测试转型的适应性

尽管自动化测试有诸多优点，但在测试转型的时候，需要明确转型的目的不是彻底抛弃手工测试，而是让测试人员从烦琐重复的机械式测试过程中解脱出来，把有限的时间和精力放到更有价值的地方，进而发现更多的产品缺陷。

需要注意的是，并不是所有的功能都能够自动化。对于某些类型的测试，例如易用性和界面友好性方面的探索型测试很难通过自动化测试进行验证，使用手工测试更能发挥测试人员的想象力，效果更好。所以，自动化测试虽然可以降低手工测试的工作量，但并不能完全取代手工测试。100%的自动化测试是不存在的，即便非常成熟的商用软件，其测试自动化率也很难超过80%。

一般而言，具有以下特点的项目比较容易实施自动化测试。

1. 需求较稳定

测试框架和脚本的稳定性决定了自动化测试的维护成本。编写测试脚本身是一个开发代码的过程，需要不断修改、调试，甚至必要的时候要修改测试框架，如果开发过程中需求变动过于频繁，则测试人员需要根据需求变化不断更新测试用例和相关脚本。也就是说，频繁的需求变化不仅会导致频繁的脚本开发和修改，也会带来高昂的成本，如果该花费高于传统的手工测试，那么自动化测试对于该项目就是不合适的。

如果项目中某些模块相对稳定，而另外一些模块需求变动比较大。建议对相对稳定的模块进行自动化测试，而更新频繁的仍坚持用手工测试，待其稳定后再进行自动化。

2. 较大型项目

自动化测试前期对需求的分析、对自动化测试框架的设计集成，进行脚本的编写与调试均需要较长的时间来完成。

一般而言，较大型的项目更容易体现自动化测试的价值。因为这类项目周期长，规模大，自动化测试在前期的框架设计和脚本编写方面的投入在后期会更加显示其优势。特别是高效的回归测试保证测试人员能够投入到新功能的验证中。如果项目较小，周期比较短，则没有足够的时间和必要去支持这样的过程，手工测试反而可能更敏捷更合适。

3. 测试人员良好的编程经验

良好的自动化测试实施离不开测试人员的工作。由于自动化测试脚本的编写实际是一个开发的过程，无论是C#、Java或JavaScript都需要测试人员对相应的编程语言和原理有一定的了解。测试人员良好的编程经验是保证自动化测试实施的基础。

4. 良好的团队合作

要保证自动化测试的成功实施，绝不是拍拍脑袋说干就一定能干好的，它不仅涉及测试工作本身流程上、组织结构上的调整与改进，甚至也包括需求、设计、开发、维护及配置管理等各个角色之间的配合，需要多部门紧密合作，包括开发、测试、运维等。如果各个不同的团队各自为政没有配合的话，一定会在实施过程中处处碰壁，既定的实施方法也无法如期开展。

8.4 测试工具的选择

目前市场上可供选择的自动化测试工具很多，各有特点。由于企业内部一般都存在许多不同种类的应用平台，所用到的开发技术也不尽相同，甚至一个应用就使用了多项技术，或同一应用的不同版本之间存在技术差异。所以选择自动化测试的工具必须深刻理解这一选择的适应性以及诸多方面的风险和成本开销。如果中途更换测试工具，那么前期的工作成果可能都付之东流，带来极大的浪费。

根据作者的经验，在企业进行自动化测试工具选型的时候，可以从以下5个方面综合考虑：

（1）尽可能多地覆盖主流操作系统和浏览器，从而降低产品投资和团队的学习成本。

（2）有良好的性价比，应充分关注产品的支持服务和售后服务的质量与水平。

（3）测试工具本身使用的技术与当前的技术发展潮流匹配，避免过于陈旧冷门的开发语言。

（4）尽量选择趋于主流成熟的产品，以便通过行业间的交流或者社区支持等方式获得更为广泛的经验分享和学习资源。

（5）良好的可扩展性，特别是对持续集成工具的支持。

市场上的自动化测试工具很多，主流的包括Rational Functional Tester（RFT）、Quick Test Professional（QTP）、TestComplete和Selenium等。表8-1列出了这4种工具的主要特点和区别。

表8-1　自动化测试工具特点比较

比 较 项	RFT	QTP	TestComplete	Selenium
编程语言	Java、C#	VBScript	VBScript、Python、Jscript、DelphiScript、C++Script、C#Script	Java、C#、Ruby、Python、Perl、PHP、JavaScript
浏览器支持	IE、Chrome、Firefox	IE、Chrome、Firefox	所有主流浏览器	所有主流浏览器
费用	付费	付费	付费	免费
操作系统	Windows、Linux	Windows	Windows	所有主流平台
工程师能力要求	较低	较低	较低	较高
技术支持	IBM	HP	SmartBear	开源社区
支持桌面程序	是	是	是	需要额外驱动进行支持

基于以上比较可以看出，RFT、QTP和TestComplete对桌面应用有着较好的原生支持并能获得官方技术支持，但在Web前端方面，Selenium无论是对浏览器和操作系统的支持还是编程语言的选择，都有着更广的覆盖面。特别是作为开源项目，免费更是Selenium的一个极具吸引力的加分项。所以，在进行前端测试工具选型的时候，如果团队具备相当的编程能力，可以考虑选择Selenium作为项目的自动化测试工具。

第9章 初识Selenium

Selenium的中文意思为“硒”，是一种非金属化学元素。硒既是工业催化剂，也是动植物必需的微量矿物质营养素。简而言之，硒摄入量不多却必不可少，往往起到魔术般的有益作用，而这也正是测试工具Selenium当年的创造者们所期望的。通过Selenium，可以将浏览器自动运行起来模拟用户操作，从而验证Web应用的行为。如今，纵观业界对Selenium的认可和火爆程度，当年的期望显然已经实现。那么是谁创造了Selenium，它又是如何发展成熟起来的呢？

本章将介绍：

- Selenium发展历史
- Selenium工具套装

9.1 Selenium发展历史

Selenium是2004年由Jason Huggins发起的一个项目[①]。当时他在ThoughtWorks参与开发和测试一个公司内部系统，该系统使用了大量的JavaScript。那时候IE还是主流浏览器，同时ThoughtWorks也使用了一些其他浏览器，特别是Mozilla系列，如果员工发现该系统在自己的浏览器中无法正常运行，就会提交缺陷。多样而复杂的使用环境让该项目进展非常缓慢，为了不让自己的时间浪费在无聊的重复性工作中，Jason希望有更好的方案提升测试效率。但当时的测试工具主要关注IE浏览器，无法充分满足该系统的所有使用环境。另一方面，购买商业工具授权的成本也会耗尽这个小型内部项目的有限预算。

① SeleniumHQ. Selenium History[OL]. [2016]. http://www.seleniumhq.org/about/history.jsp.

Jason考虑到该系统所有用到的浏览器都支持JavaScript，基于这一点，他带领团队创造性地研发了一个JavaScript程序库与页面进行交互，通过这个程序库实现对多个浏览器的测试自动化。

该代码库最初被称为“Selenium”，也就是后来的“Selenium Core”，于2004年基于Apache 2授权发布。Selenium Core也是后来出现的Selenium Remote Control和Selenium IDE的核心技术。

因为Selenium当时使用JavaScript编写，它需要开发人员把需要测试的应用、Selenium Core和测试脚本部署到同一台服务器上以避免违反浏览器的安全规则和JavaScript沙箱策略。但是在后续的开发过程中，Jason发现并不是总能满足这种要求。为了解决这个问题，他们进一步编写了一个HTTP代理，这样测试脚本可以将页面操作指令通过HTTP请求发送到代理上，连接协议则基于Selenium Core的语法建模，称之为“Selenese”。这套技术实现被统称为“Selenium Remote Control”，简称为“Selenium RC”或“Selenium 1”。Selenium RC的诞生是划时代的，满足了工程师用多种编程语言开发控制脚本的需求。实际上任何语言只要支持发送HTTP请求就可以支持Selenium RC。

但另一方面，Selenium RC也有明显的局限性。由于它使用JavaScript驱动页面，这导致其无法突破JavaScript的沙箱限制，很多复杂的测试场景难以实现。尽管Selenium RC当时已经取得了广大社区的认可，很多公司都投入了实施，但伴随着互联网相关技术的不断发展，这一局限变得日益突出，例如Google作为Selenium的重度用户，却总被限制在浏览器的有限操控范围内，无法有效充分地对其服务进行高覆盖率的测试。2006年Google工程师Simon Stewart发起了一个叫WebDriver的项目。此项目通过让测试引擎直接调用浏览器的原生API来绕过JavaScript的沙箱限制，代价是针对不同的浏览器需要分别开发驱动。Selenium WebDriver于2007年发布并很快开始支持IE和Firefox。

尽管WebDriver取得了很大的成功，但它较Selenium RC也有明显的不足，就是RC基于HTTP代理的架构提供了广泛的编程语言支持，而WebDriver仅支持Java。为了充分融合两者的优点，并吸纳开源社区的贡献者继续合作，这两个项目于2008年合并，取名为Selenium 2。

Selenium 2既向下兼容Selenium RC的已有功能，也提供了更现代化的WebDriver API来满足更复杂的测试需求。对于新项目框架的选型，推荐基于WebDriver来搭建自动化测试框架。

9.2 Selenium工具套装

随着技术的不断进步，当前的Selenium家族已经发展成为一套强大的开源测试工具，除了上节提到的Selenium RC和Selenium WebDriver，还包括Selenium IDE和Selenium Grid，如图9-1所示。它们从不同方面满足了用户的测试需求，让Selenium发展成为测试领域里实实在在的“硒”。接下来，本书将针对这些工具的特点逐一展开分析。

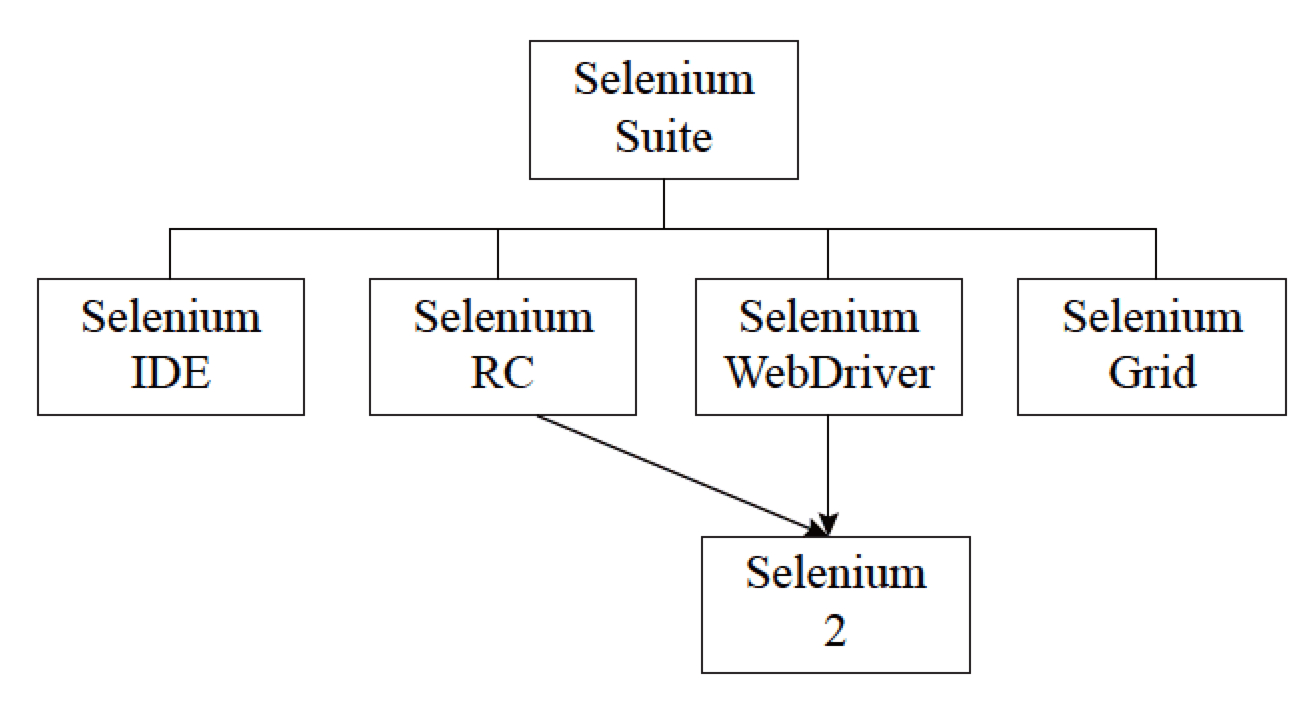


图9-1 Selenium 工具套装

9.2.1 Selenium RC

Selenium RC的核心组件由Selenium Core和Remote Control Server构成，如图9-2所示。Selenium Core本身是一个JavaScript的类库，包含了操作浏览器的相应代码。运行测试用例时，测试代码会通过封装好的客户端API将浏览器操作指令通过HTTP请求发送到Remote Control Server。Remote Control Server接收到HTTP请求后，根据指令访问被测页面获取对应网页数据，在获取的网页数据里注入Selenium Core的代码，然后由Selenium Core根据测试指令操作浏览器，进行页面操作。

在这个流程里，Remote Control Server的存在不仅仅是为了支持多种客户端编程语言，更关键的是发挥了一个HTTP代理的角色。大家知道，现在主流的浏览器出于安全考虑，都支持JavaScript的同源策略[①]，它是浏览器最核心也是最基本的安全功能。在浏览器里，某域名下的JavaScript语句无法获取其他域名下的数据或操作其他域名下的对象，所以外部引用的代码因为域名不同会被拒绝执行。为了避免作为外部引用的Selenium Core被

① Mozilla. CORS[OL]. [2016]. https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS.

浏览器拒绝执行，Remote Control Server担当了HTTP代理的角色。它在收到网页数据后注入Selenium Core并重组网页数据，然后再将重组的数据转发给浏览器。这样，浏览器无法意识到Selenium Core实际上是注入的外部代码，从而成功绕过了浏览器同源策略的限制。不得不说这的确是一个非常精妙的创新。

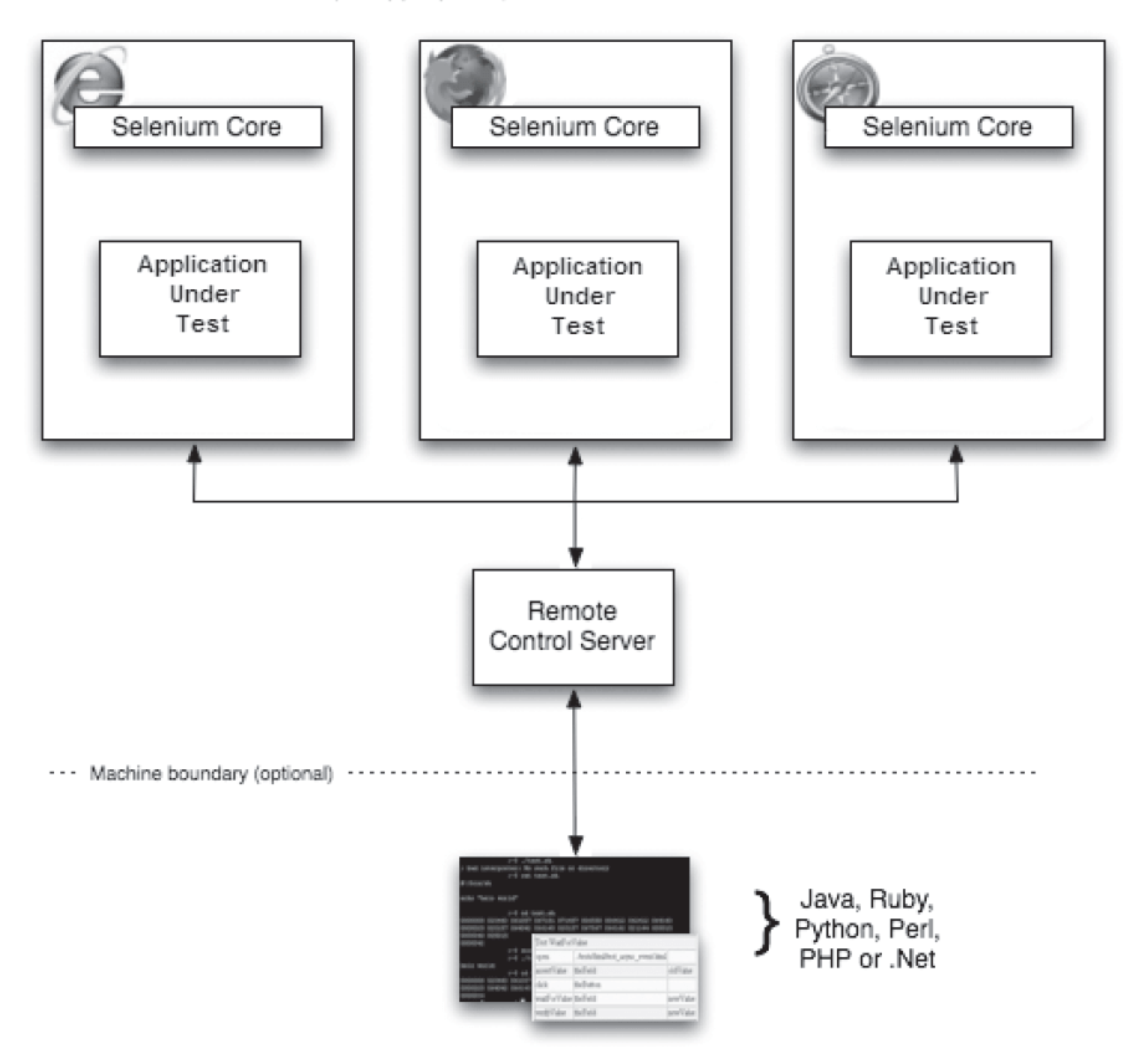


图9-2 Selenium RC工作原理

9.2.2 Selenium WebDriver

Selenium RC通过注入Selenium Core，使用相同的JavaScript类库来操作浏览器，因此无法兼顾到不同浏览器的差异性；同时由于JavaScript的沙箱属性，有些复杂功能难以实现。WebDriver[①]的实现原理完全不同，它直接利用浏览器的接口进行网页操作，更

① Simon Stewart. Introducing WebDriver[OL]. 2009. https://opensource.googleblog.com/2009/05/introducing-webdriver.html.

接近用户使用的真实场景，更简洁的面向对象的客户端接口也大大提高了脚本的编写效率。

读者可能会有一个疑问，既然如此，那Remote Control Server还有存在的必要吗？关于这个问题，请读者谨记以下几点，它们也充分体现了RC与WebDriver的合并理念：

（1）为避免歧义，现在Remote Control Server成为了Selenium Server功能的一部分。

（2）Selenium Server考虑到兼容性，同时支持Selenium RC和Selenium WebDriver。

（3）Selenium WebDriver测试程序既可以通过Selenium Server操作浏览器，也可以直接调用浏览器来驱动。

9.2.3 Selenium Grid

在第1.3节关于手工测试的局限性中，提到过针对当前多样化的操作系统和浏览器，为了保证产品的高质量需求，要让测试覆盖到尽可能多的环境，Selenium Grid就是为满足这种分布式测试需求而诞生的。本书将在第14章详细介绍Selenium Grid的配置与使用。

9.2.4 Selenium IDE

2006年，日本工程师Shinya Kasatani对Selenium产生了浓厚兴趣，并成功地在Firefox里开发了一个具有交互式界面的插件，可以支持脚本的录制、播放、修改和导出等功能。后来，该项目集成到了Selenium套件内，演变为Selenium IDE（Integrated Development Environment），它小巧简便，容易上手，对初次接触自动化测试的人员熟悉Selenium、了解脚本编写很有帮助。

1. 安装Selenium IDE

由于Selenium IDE是Firefox的一个插件，可以从Firefox插件管理器里直接安装获取，具体步骤如下:

（1）打开Firefox浏览器，单击右上角按钮，再单击“Add-ons”按钮，如图9-3所示。

（2）在插件管理器中搜索Selenium IDE，并按Most Users排序，可以方便地找到Selenium IDE插件，如图9-4所示。单击Add to Firefox按钮即可完成安装并重启Firefox。

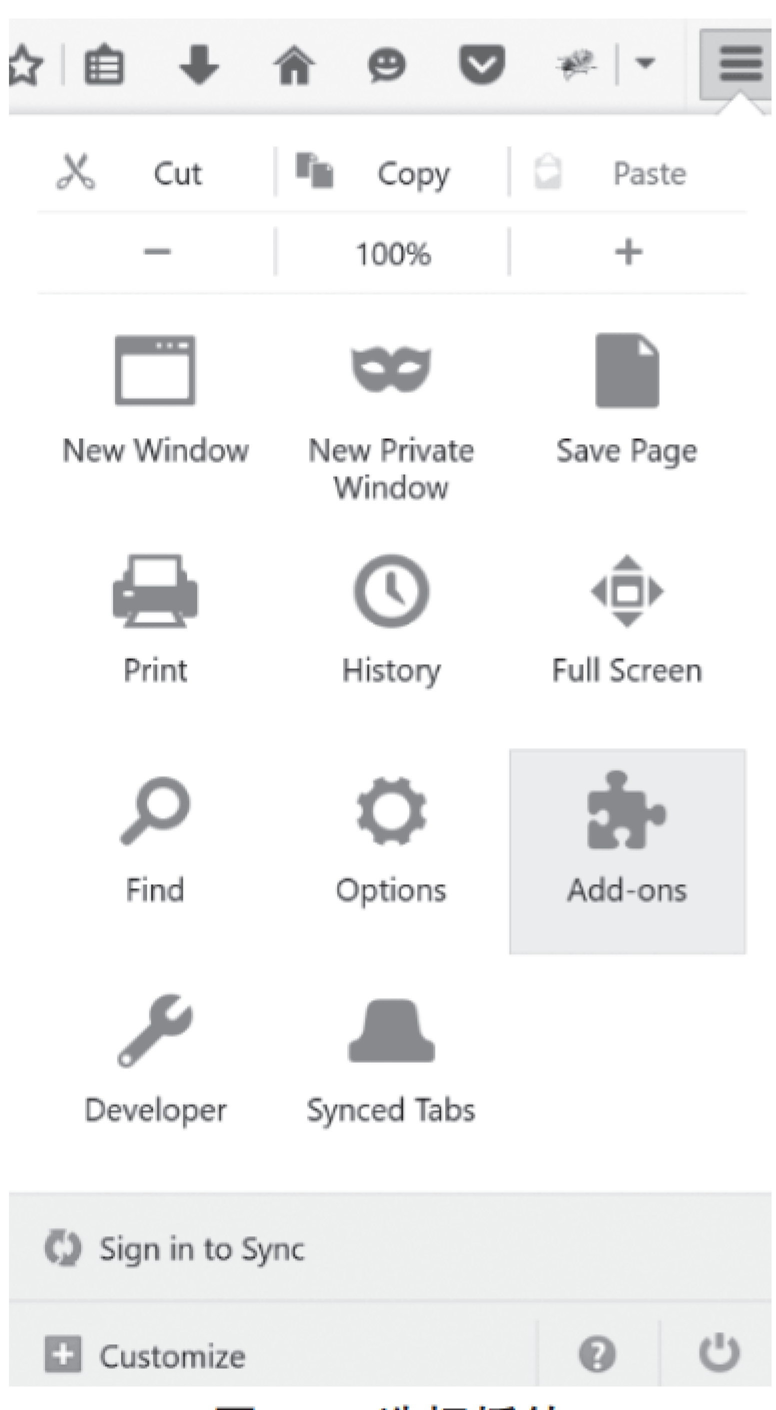


图9-3　选择插件

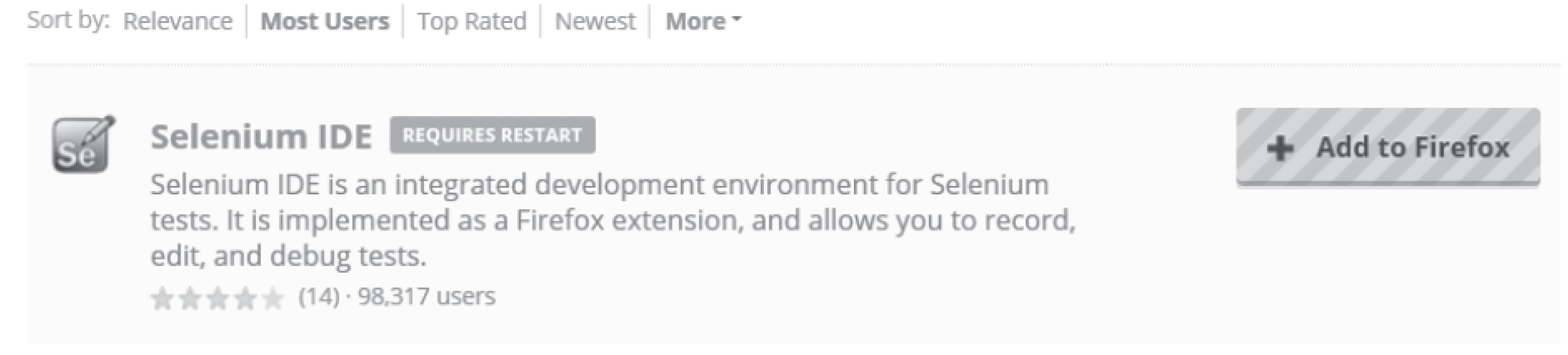


图9-4　安装Selenium IDE

2. 脚本录制与播放

安装完Selenium IDE后，就可以在Firefox的工具菜单下找到IDE的启动项，如图9-5所示。

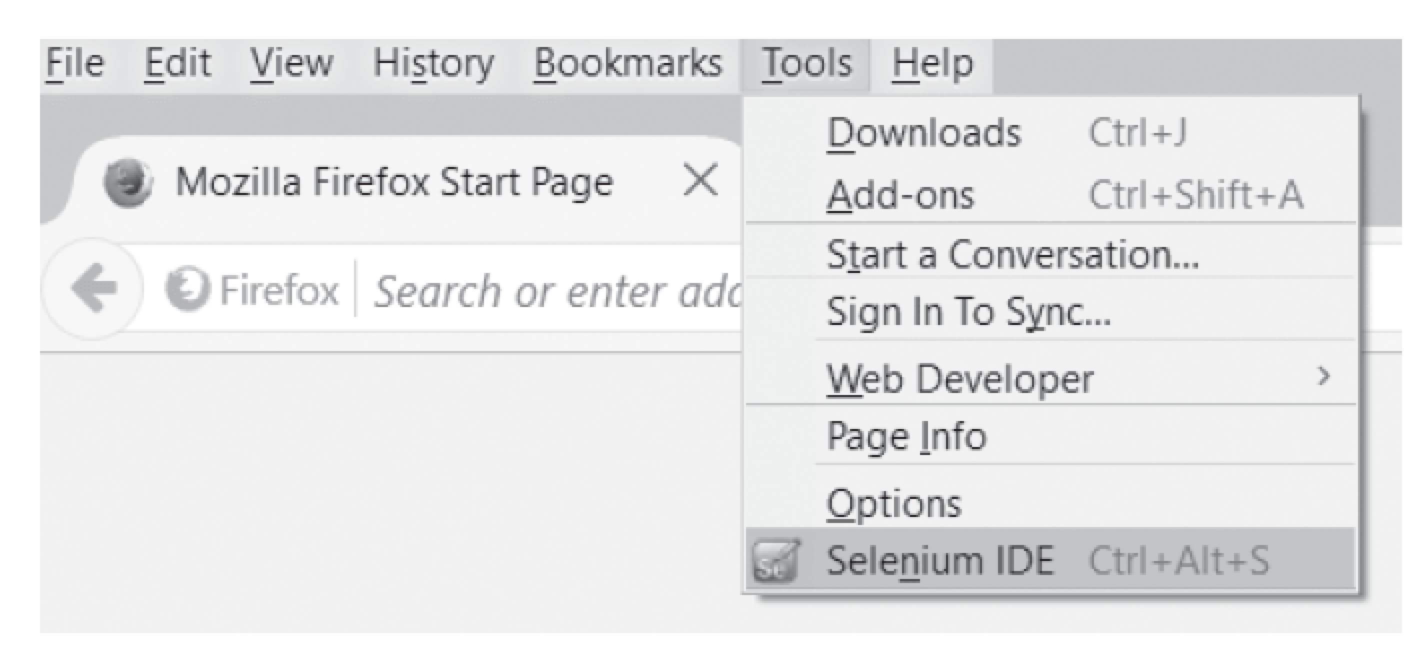


图9-5　启动Selenium IDE

选择Selenium IDE菜单项会弹出Selenium窗口，该窗口包含录制按钮、脚本显示区、

脚本编辑区等多块区域，如图9-6所示。

图9-6 Selenium IDE 界面

接下来以必应搜索为例演示如何进行脚本录制、添加验证条件以及进行脚本回放，具体步骤如下：

（1）打开Firefox浏览器和Selenium IDE。录制按钮默认为启动状态，即表明录制已经开始。

（2）切换到Firefox，在浏览器地址栏中输入https://www.bing.com/。

（3）必应首页加载完成后，在搜索输入框中输入selenium，单击搜索按钮。

（4）回到Selenium IDE窗口，单击录制按钮，停止录制，即可看到脚本显示区已经有了录制好的脚本命令。

（5）在脚本播放工具条中单击播放按钮，回放刚录制好的脚本，图9-7显示测试成功通过。

（6）上一步使用回放功能，通过执行脚本重复了人工所做的搜索过程。但是该脚本只是操作浏览器，还没有验证结果。请注意搜索完成后必应页面的标题显示为selenium – Bing，接下来为脚本加入验证逻辑，以检查标题是否正确。单击脚本显示区的空白部分后，脚本编辑区变为可编辑状态。

（7）在Command下拉框选择assertTitle，在Target一栏输入selenium – Bing。再次回放脚本，图9-8显示页面标题为验证期待的selenium-Bing。

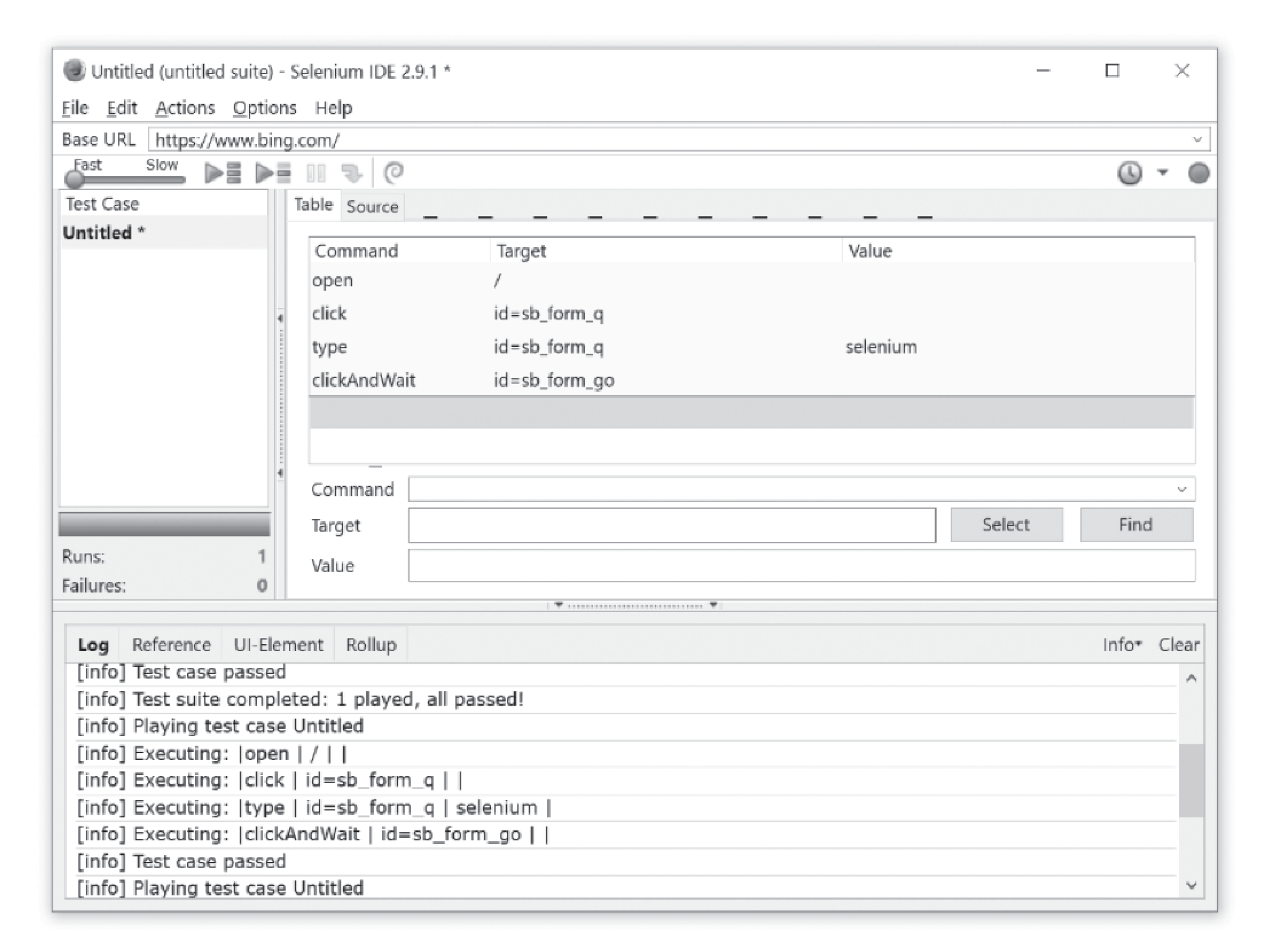


图9-7　完成录制并通过测试

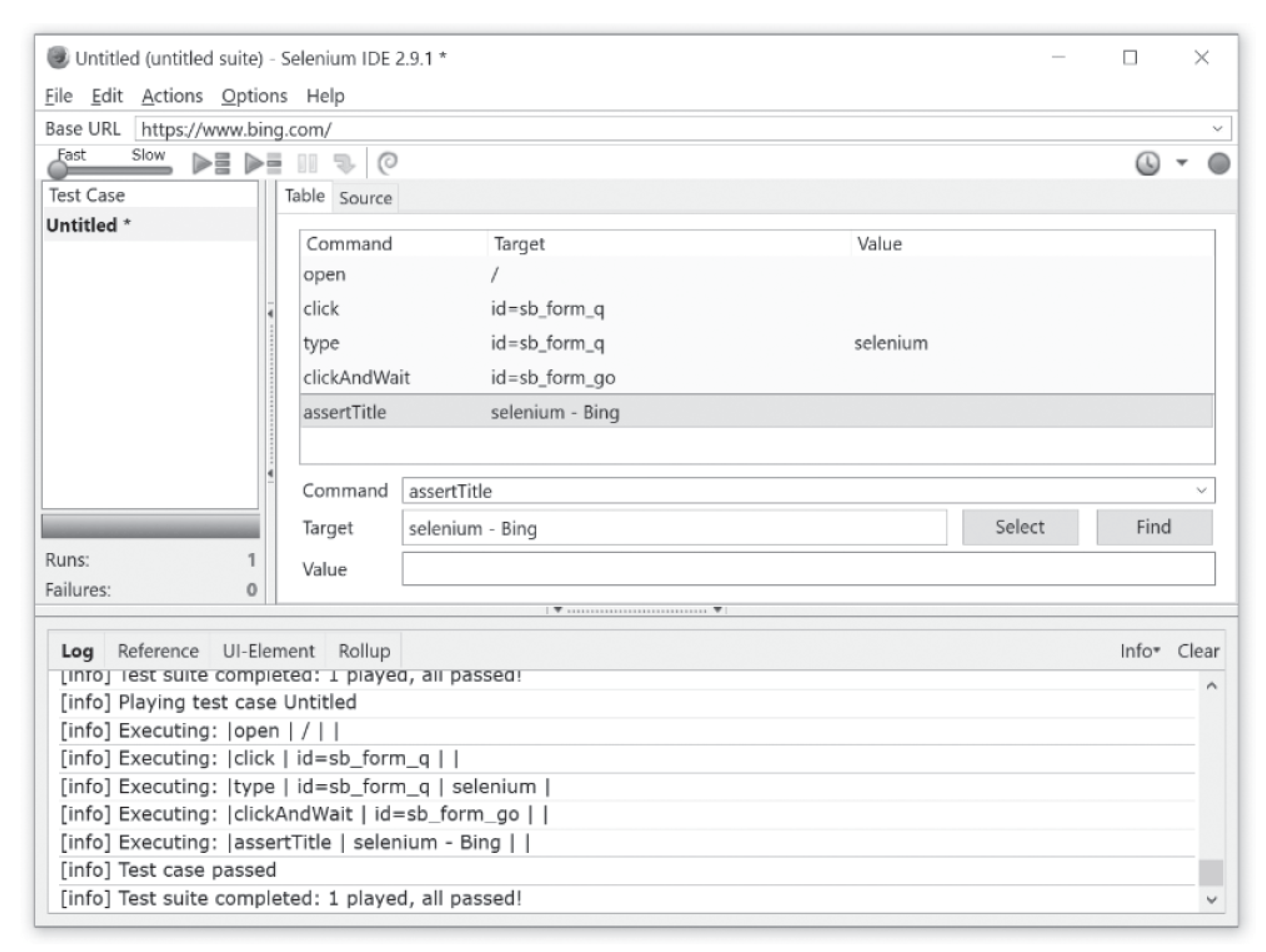


图9-8　验证页面标题，脚本运行成功

（8）在Selenium IDE里把Target一栏改为selenium，再次回放脚本，测试日志区将显示出现错误，如图9-9所示。错误的原因是页面标题不匹配。

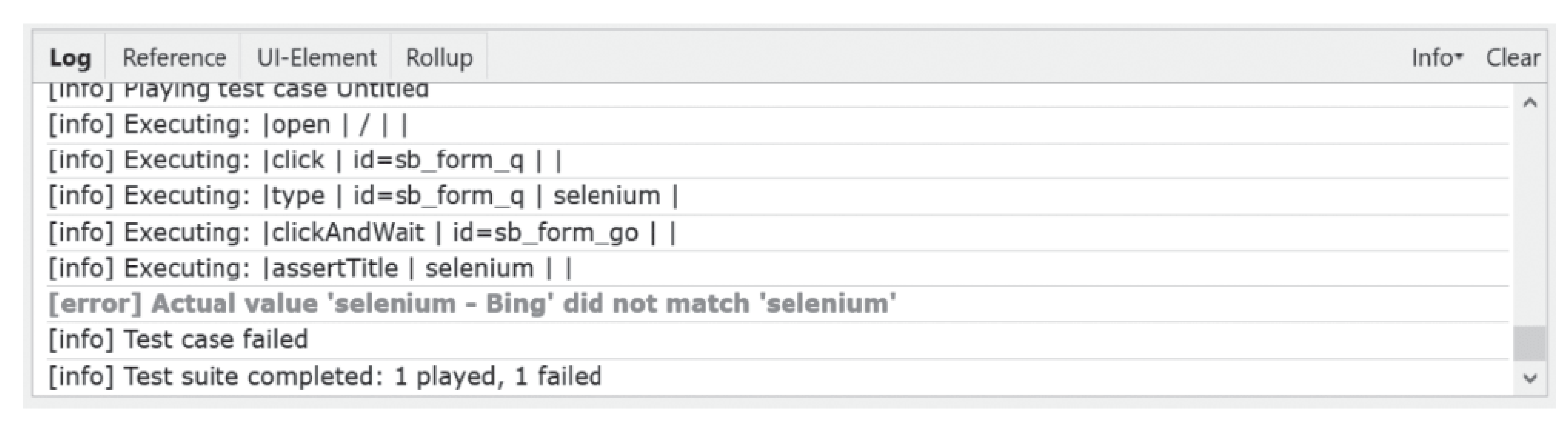


图9-9　日志显示有错误

以上是最简单的脚本录制与修改实例。Selenium IDE基于Selenese在Command下拉框提供了丰富的页面操作和验证方法，读者可以基于此实例进一步尝试和了解在Selenium IDE内录制和修改脚本的神奇之处。

3. 导出脚本

Selenium IDE另一个强大的功能是能够把录制好的脚本导出为其他编程语言的测试代码。这个功能极大地拓展了Selenium IDE的实用性，使不同背景的测试人员都可以方便地选择自己熟悉的编程语言来进一步研究。在实践过程中，开发人员可以先在Selenium IDE中进行脚本录制和修改，通过实时回放调整好参数后再将其导出并纳入到自己的自动化测试系统内。

从Selenium中导出脚本非常简单，如图9-10所示，只需要在Selenium IDE里选择File菜单，单击Export Test Case As菜单项即可。Selenium IDE支持导出多种主流编程语言和测试框架，在本例中，选择的是C#/NUnit/WebDriver，即导出基于C#语言，由NUnit单元测试框架组织，使用Selenium WebDriver API的测试代码。

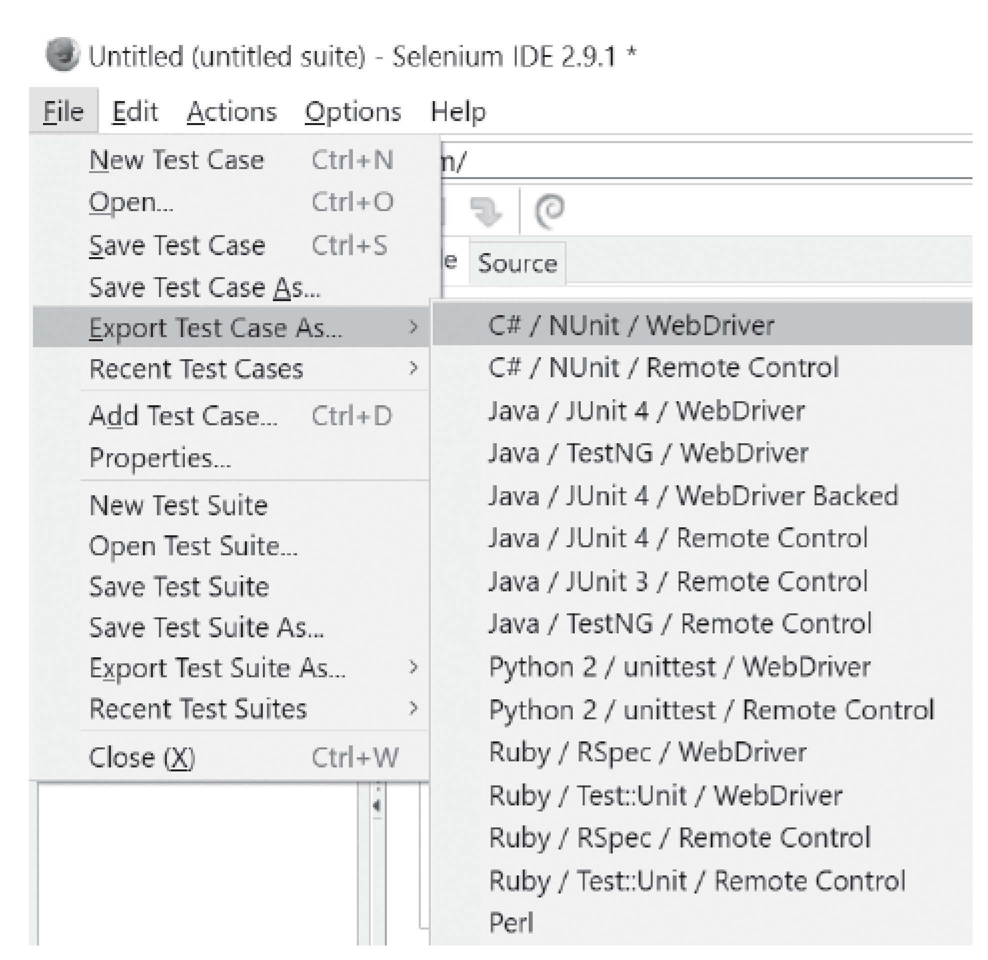
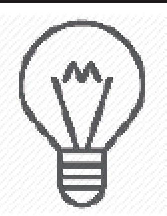


图9-10　导出测试脚本



使用Selenium IDE导出的测试代码有时会出现少量问题，需要测试人员调试修改。

以下为导出的核心测试代码：

```
private IWebDriver driver;

private StringBuilder verificationErrors;

private string baseURL;

private bool acceptNextAlert = true;

[SetUp]

public void SetupTest()

{

    driver = new FirefoxDriver();

    baseURL = "https://www.bing.com/";

    verificationErrors = new StringBuilder();

}

[TearDown]

public void TeardownTest()

{

    try

    {

        driver.Quit();

    }

    catch (Exception)

    {

        // Ignore errors if unable to close the browser

    }

    Assert.AreEqual("", verificationErrors.ToString());

}


[Test]
```

```
public void The1Test()
{
    driver.Navigate().GoToUrl(baseURL + "/");
    driver.FindElement(By.Id("sb_form_q")).Click();
    driver.FindElement(By.Id("sb_form_q")).Clear();
    driver.FindElement(By.Id("sb_form_q")).SendKeys("selenium");
    driver.FindElement(By.Id("sb_form_go")).Click();
    Assert.AreEqual("selenium - Bing", driver.Title);
}
```

尽管Selenium IDE功能强大，易于上手，但由于测试人员无法控制其脚本的录制逻辑，因此对于复杂或者AngularJS的页面，录制的脚本往往难以维护。故Selenium IDE主要适合让测试人员快速上手以及进行原型方案的设计。下一章将用纯编程的方式构建测试用例。

第10章 Selenium WebDriver与元素定位

Selenium WebDriver可以由多种编程语言驱动，其中，基于C#和NUnit、Java和TestNG的测试方案已经非常成熟并被广泛使用。

本章将介绍：

- 搭建集成开发环境
- NUnit单元测试框架
- 编写测试用例
- 使用工厂模式创建驱动对象
- 定位页面元素

10.1 搭建集成开发环境

以下为使用微软Visual Studio搭建Selenium WebDriver测试环境的步骤：

（1）启动Visual Studio 2015，选择File→New→Project命令。

（2）在弹出的New Project对话框中选择Visual C#模板，单击Unit Test Project选项，修改项目名称为WebDriverTest，单击OK按钮，如图10-1所示。

（3）在Solution Explorer里右击WebDriverTest项目，选择References→Manage NuGet Packages命令。

（4）.Net平台提供了丰富的Selenium客户端类库，Selenium.WebDriver和Selenium.Support是最基本也是最常用的，其中前者提供了对浏览器驱动的绑定，后者对HTML复杂元素提供了定位帮助类。如图10-2所示，在NuGet：WebDriverTest的Browse页面里搜索selenium关键字，安装Selenium.WebDriver和Selenium.Support。

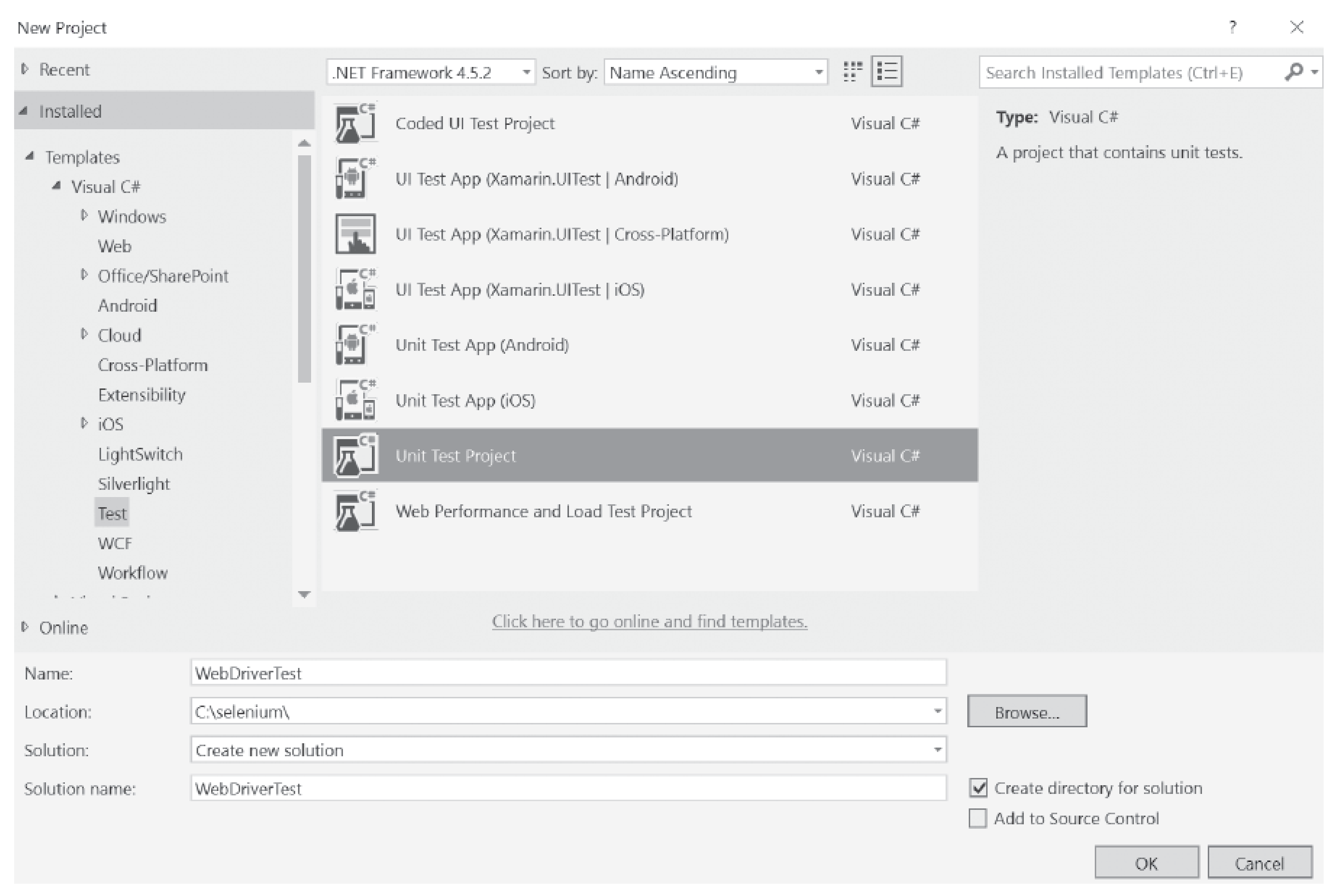


图10-1　创建测试项目

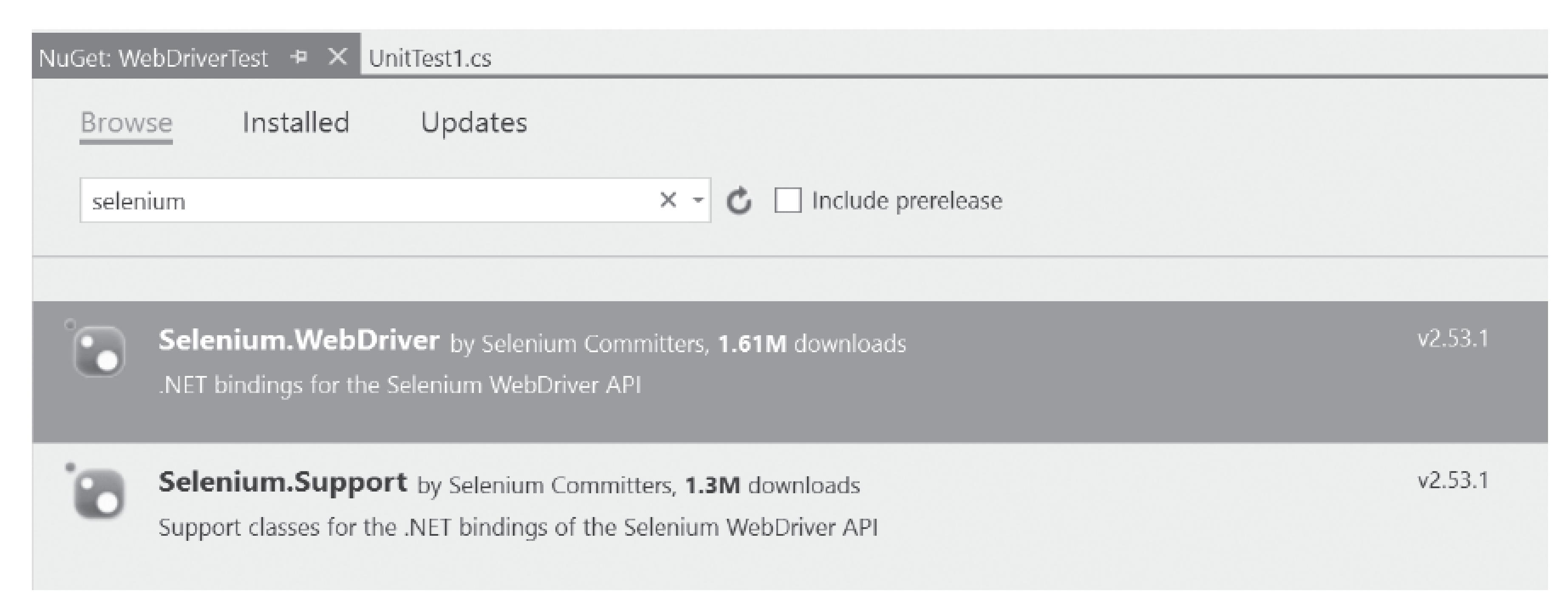


图10-2　安装Selenium支持库

（5）在NuGet：WebDriverTest管理器中下载Chrome驱动，如图10-3所示。关于如何支持其他浏览器，以及对应驱动的安装，将在12.4节详细介绍。

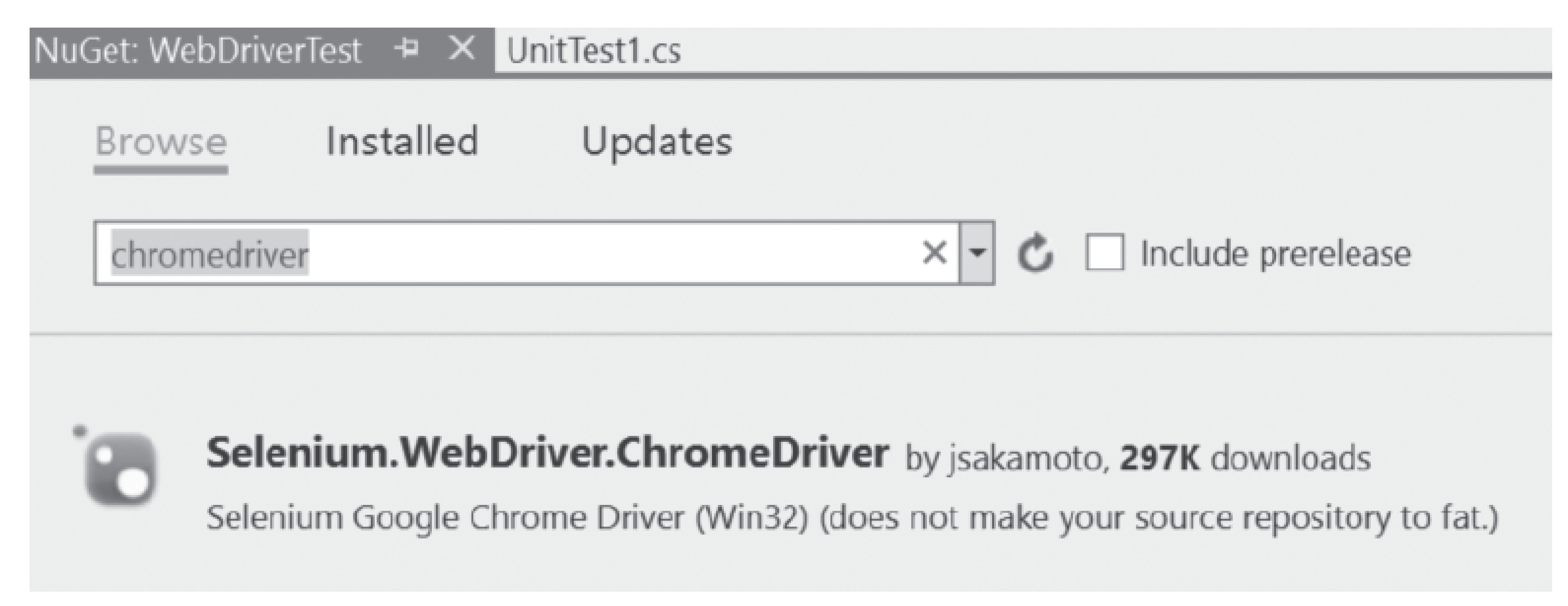


图10-3　添加驱动

（6）打开UnitTest1.cs文件，添加以下代码引用后，针对Chrome进行WebDriver开发的环境配置工作即完成。

```
using OpenQA.Selenium;
using OpenQA.Selenium.Chrome;
```

10.2　NUnit测试框架

任何测试都需要一个框架组织测试用例，自动化测试也不例外。NUnit是目前.Net平台上最成熟、使用最广泛的测试框架，它属于xUnit家族的一员，最初由Kent Beck、Erich Gamma等共同开发完成。在作者编写此书时，最新的NUnit版本为3.4.1。本节将使用NUnit构建自动化测试框架。

在C#工程内使用NUnit，同样可以通过NuGet：WebDriverTest添加项目引用。如图10-4所示，打开NuGet浏览页面后，搜索NUnit，下载NUnit类库和NUnit3TestAdapter。NUnit3TestAdapter是一个Visual Studio插件，提供了在Visual Studio集成开发环境内对NUnit测试用例进行可视化操作的功能。

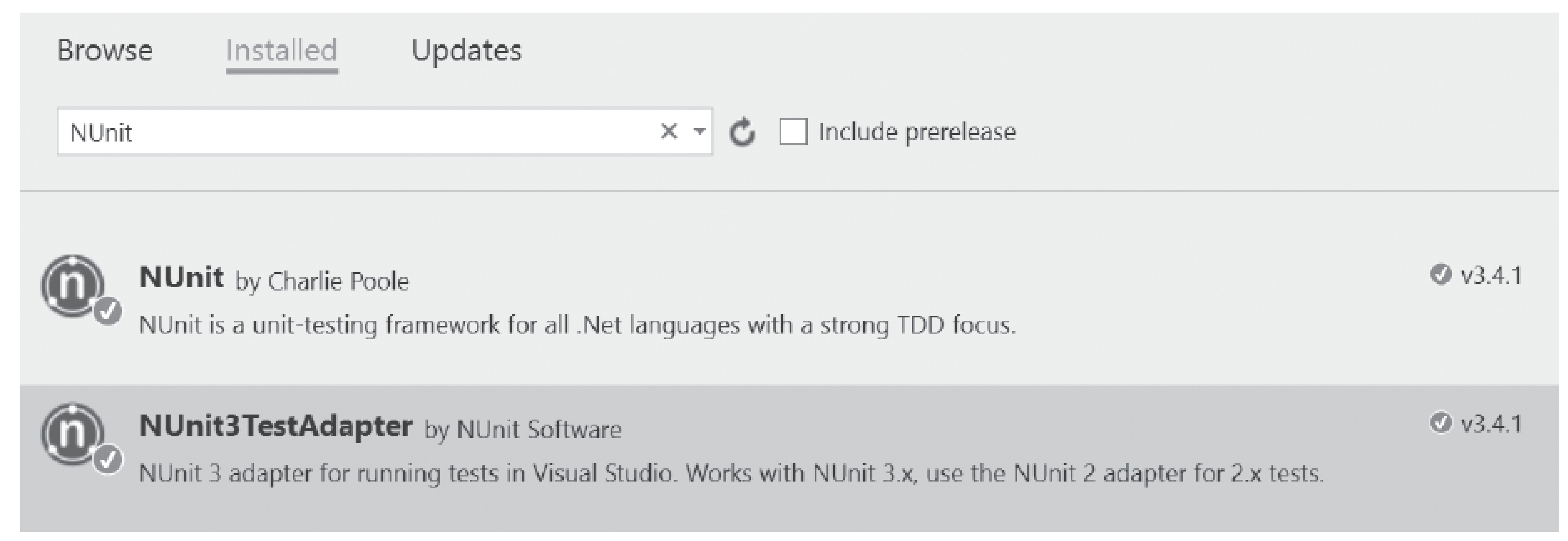


图10-4　安装NUnit

NUnit提供了多个属性对测试用例进行组织，如表10-1所示。

表10-1　NUnit属性

属性名称	描　　述
TestFixture	把某个类指定为一组测试用例的集合
Test	把某个方法指定为一个测试用例
Setup	在每个测试用例执行之前进行测试环境初始化
TearDown	在每个测试用例执行完成后进行测试环境清除
TestFixtureSetup	执行耗时且可以被多个用例共享的初始化代码
TestFixtureTearDown	执行耗时且可以被多个用例共享的环境清除代码

NUnit允许在测试执行过程中对测试状态进行断言，辅助判断测试成功与否。表10-2是常用的断言列表。

表10-2 NUnit断言

断言名称	描 述
Assert.AreEqual	判断两个对象是否相同
Assert.AreNotEqual	判断两个对象是否不同
Assert.AreSame	判断是否引用了相同对象
Assert.AreNotSame	判断是否引用了不同对象
Assert.Contains	判断一个对象是否属于某个数组
Assert.Greater	判断是否大于
Assert.Less	判断是否小于
Assert.IsTrue	判断是否为true
Assert.IsNull	判断是否为null
Assert.IsEmpty	判断一个字串或数组是否为空

在UnitTest1.cs文件里添加对NUnit.Framework命名空间的引用，并按如下修改代码完成对测试用例的组织。

```
namespace WebDriverTest
{
   [TestFixture]
   public class BingTest
   {
      [SetUp]
      public void SetUp()
      {
      }
      [TearDown]
      public void TearDown()
      {
      }
      [Test]
```

```
        public void Search()
        {
        }
    }
}
```

10.3　编写测试用例

Selenium IDE能够自动跟踪用户的鼠标事件、定位点击的HTML元素、录制脚本。那么当脱离Selenium IDE后该如何获得类似的功能呢？本节仍以必应搜索为例，来演示如何用C#编写相同功能的测试用例。

首先需要对页面元素进行定位，操作步骤如下：

（1）启动Chrome，在地址栏输入http://www.bing.com，按回车键。

（2）按F12键，启动Chrome调试窗口，单击左侧的Select an element in the page to inspect it按钮后，单击页面上的搜索框，此时可以看到搜索框的id为sb_form_q，如图10-5所示。



图10-5　选择搜索框

（3）使用同样的方法得到搜索按钮的id为sb_form_go。

（4）修改代码，创建Chrome驱动对象并基于已得到的id值对搜索框和搜索按钮进行定位和操作。修改后的代码如下。

```
[TestFixture]
public class BingTest
```

```
{
   private IWebDriver driver = null;
   [SetUp]
   public void SetUp()
   {
      driver = new ChromeDriver(); // construct the ChromeDriver object
   }
   [TearDown]
   public void TearDown()
   {
      driver.Quit();
   }
   [Test]
   public void Search()
   {
      // navigate to bing.com
      driver.Navigate().GoToUrl("http://www.bing.com");
      // locate the search box by id, set "selenium" as search text
      driver.FindElement(By.Id("sb_form_q")).SendKeys("selenium");
      // locate the search button and click
      driver.FindElement(By.Id("sb_form_go")).Click();
   }
}
```

以上代码展示了如何通过By.Id，基于元素id定位到提交按钮。

必应在完成搜索后，会返回若干个相关的搜索结果。假设本次测试用例是检查Selenium的官方网站是否在首页被搜索到，是否还可以用id进行元素定位呢？

（1）再次在Chrome中通过按F12键选择列表中的Selenium官方网站，如图10-6所示。

（2）从图10-6中可以看到对应的超链接元素并没有id属性，所以无法使用id对其进行定位。但其href属性是http://docs.seleniumhq.org，能够唯一地识别该对象。根据这一特

点，可以使用XPath对超链接元素对象进行定位，示例代码如下。关于XPath的使用，本章后续会进一步介绍。



图10-6　定位搜索结果

```
public void Search()
{
   driver.Navigate().GoToUrl("http://www.bing.com");
   driver.FindElement(By.Id("sb_form_q")).SendKeys("selenium");
   driver.FindElement(By.Id("sb_form_go")).Click();
   System.Threading.Thread.Sleep(5000);
   Assert.IsNotNull(driver.FindElement(By.XPath("//a[@href='http://docs.seleniumhq.
org/']")));
}
```

请注意，以上代码在执行断言之前通过调用System.Threading.Thread.Sleep等待了5秒，原因是需要确保搜索完成而且结果已经返回给了浏览器。本书将在后续章节详细介绍多种等待方式并比较其适用性。

（3）在Visual Studio的主菜单，选择Test→Windows→Test Explorer命令，即可以看到创建好的Search用例。右击Search，选择Run Selected Tests后运行测试用例。图10-7显示出测试已通过。

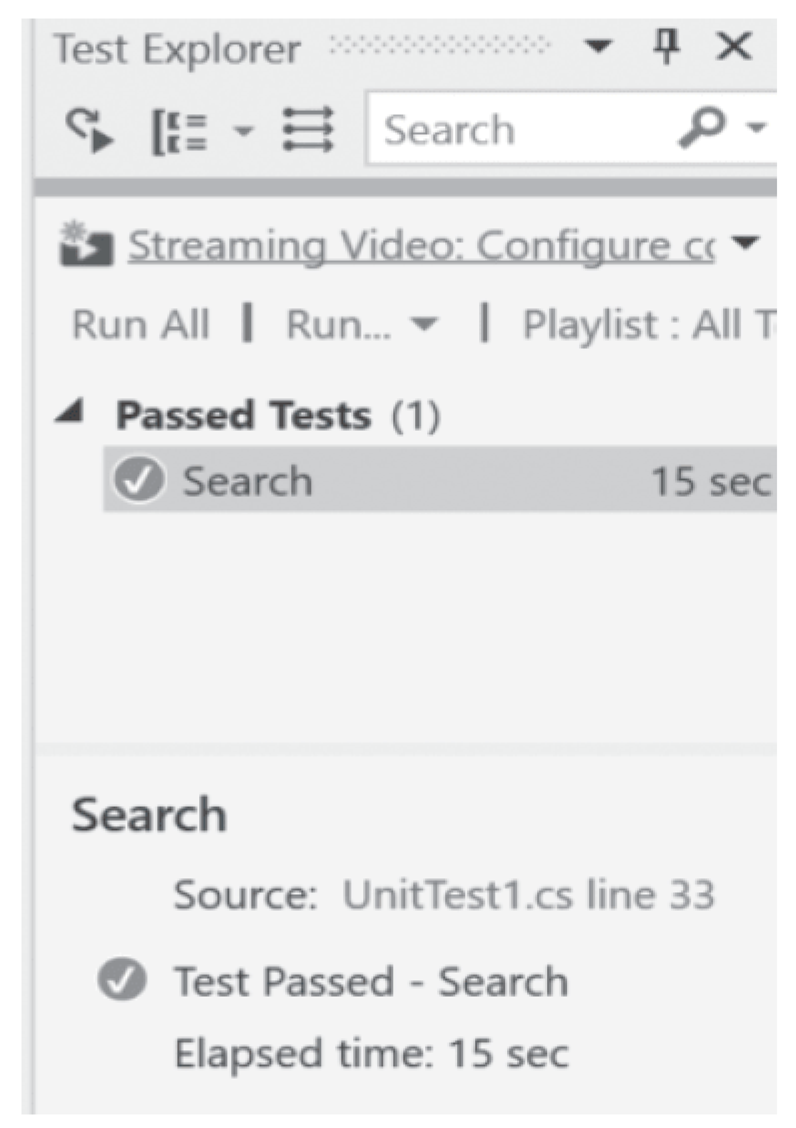


图10-7　运行测试用例

10.4　使用工厂模式创建驱动对象

在上一节示例代码中，浏览器驱动对象是通过调用驱动类对应的构造函数来创建的，但直接在测试用例代码中创建浏览器驱动对象需要显式指定驱动器类型，对于多浏览器测试灵活性不够。如果在实践中需要让相同的测试用例在多种浏览器里进行测试，该如何指定浏览器类型呢？难道每个测试用例都用不同浏览器驱动重写一遍吗？

针对这个需求，可以结合配置文件和工厂模式动态指定浏览器对象的类型。由于配置文件不会被编译到生成的模块内，因此当临时添加或删除浏览器测试类型时，无需重新编译项目。

以下为app.config的示例代码：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="homeaddress" value="http://www.bing.com"/>
    <add key="browser" value="Chrome"/>
```

```
  </appSettings>
</configuration>
```

以下为通过工厂模式创建浏览器驱动对象的示例代码：

```
using OpenQA.Selenium;
using OpenQA.Selenium.Chrome;
using OpenQA.Selenium.Firefox;
using OpenQA.Selenium.IE;
using System.Configuration;
namespace WebDriverTest
{
    class DriverFactory
    {
        public static IWebDriver InitWebDriver()
        {
            IWebDriver driver = null;
            AppSettingsReader appreader = new AppSettingsReader();
            string browser = (string)appreader.GetValue("browser", typeof(string));
            switch(browser)
            {
                case "Chrome":
                        driver = new ChromeDriver();
                        break;
                case "Firefox":
                        driver = new FirefoxDriver();
                        break;
                default:
                        driver = new InternetExplorerDriver();
                        break;
            }
```

```
            driver.Url = (string)appreader.GetValue("homeaddress", typeof(string));
            return driver;
        }
    }
}
```

10.5　定位页面元素

从上一节的示例可以看到，无论是填写表单，单击按钮，还是断言元素是否存在，一个前提是需要先对HTML页面里的元素进行定位，否则后续的DOM操作也就无从谈起。WebDriver提供了FindElement和FindElements函数用于定位一个或多个页面元素。

1. FindElement

该函数的功能如下：

- 如果没有找到任何符合查找条件的元素，则抛出NoSuchElementException异常。
- 如果找到一个相符元素，则返回对应的IWebElement对象。
- 如果找到多个相符元素，则返回第一个IWebElement对象。

2. FindElements

该函数的功能如下：

- 如果没有找到任何符合查找条件的元素，则返回一个列表，里面元素为空。
- 如果找到一个相符元素，则返回包含一个IWebElement对象的列表。
- 如果找到多个相符元素，则返回包含多个IWebElement对象的列表。

在前面的例子里已经尝试了使用id和XPath进行元素定位，本节将继续对Selenium WebDriver提供的定位方法和最佳实践做详尽解释。

10.5.1　基于id定位

通过id进行元素定位是执行效率最高的识别策略，在前端工程师设计页面的时候，建议为页面内所有的关键元素都加上id，以此提高核心功能的可测试性和性能，有效降低编写自动化测试脚本的难度。

被测HTML代码如下：

```
<html>
   <body>
      <div>
         <input type="text" id="txt_input" name="txt_input">
         <input type="button" id="btn_submit" value="Submit">
      </div>
   </body>
</html>
```

测试代码如下：

```
IWebElement txt_input = driver.FindElement(By.Id("txt_input"));
IWebElement btn_submit = driver.FindElement(By.Id("btn_submit"));
txt_input.SendKeys("automation");
txt_input.getText("automation");
btn_submit.Click();
```

在以上代码中，被测HTML文件内的两个input元素有id值，可以通过By.Id指明使用id作为定位手段，调用FindElement后获取得到对应的元素对象。SendKeys和Click是IWebElement的常用方法，在获得了元素对象后，可以用来设置对象文本和模拟鼠标单击动作。尽管id是一种高效的定位手段，但某些页面存在元素没有id、有重复id，或者id由开发框架随机生成的情况，id存在使用局限性。

10.5.2　基于Name定位

基于Name的定位也是常用的方式，特别是表单里的控件都会设置Name，其值在提交表单后可以被服务器获得。与id类似，基于Name的定位也有较好的测试性能，但如果页面中多个元素有相同的Name，则需要通过FindElements先找到所有符合条件的元素，然后进一步定位。

被测HTML代码如下：

```
<html>
   <body>
      <div>
         <input type="text" id="txt_input" name="txt_input">
         <input type="button" id="btn_submit" value="Submit">
      </div>
   </body>
</html>
```

测试代码如下：

```
IWebElement txt_input = driver.FindElement(By.Name("txt_input"));
```

10.5.3 基于ClassName定位

根据class属性，可以使用By.ClassName定位页面元素。由于相同的class可能被添加到多个页面元素，因此该方法比较适合于页面中使用了独特class属性的元素。

被测HTML代码如下：

```
<html>
   <head>
      <style type="text/css">
         .entry-title{
            word-wrap:break-word;
            font-size:45px;
         }
      </style>
   </head>
   <body>
      <header>
         <h1 class="entry-title">This is the Header</h1>
```

```
        </header>
    </body>
</html>
```

测试代码如下：

```
IWebElement txt_header = driver.FindElement(By.ClassName("entry-title"));
```

10.5.4　基于TagName定位

通过By.TagName可以对页面中的某种对象进行查找。由于一个页面中具有相同TagName的元素可能有多个，这种定位方式比较适合的场景是用FindElements查找到该类的所有对象后，对其进行计数和修改等操作。

被测HTML代码如下：

```
<html>
    <body>
        <div>
            <input type="text">
            <input type="text">
            <button>Click</button>
        </div>
    </body>
</html>
```

测试代码如下：

```
ReadOnlyCollection<IWebElement> elements = driver.FindElements(By.TagName("input"));
```

10.5.5　基于LinkText定位

超链接是构成一个网站应用的重要元素，WebDriver专门提供了By.LinkText基于超链接的显示文字查找anchor元素。在以下示例中，超链接的文本为“Go - Microsoft

Development Network”，把其作为参数传给By.LinkText即可找到对应的IWebElement对象。

被测HTML代码如下：

```
<html>
   <body>
      <div>
         <a href="https://msdn.microsoft.com/en-us/default.aspx">
            <strong>
               <i>Go</i>
            </strong>
            <i>
               -
               <i>Microsoft</i>
               <i>Development</i>
               <i>Network</i>
            </i>
         </a>
      </div>
   </body>
</html>
```

测试代码如下：

```
IWebElement element = driver.FindElement(By.LinkText("Go - Microsoft Development Network"));
```

10.5.6 基于PartialLinkText定位

基于LinkText的定位手段需要完全匹配显示的文字，对于某些动态页面或者是支持多语言的网站，链接的一部分文字可能是稳定的，但另一部分会发生变化，在这种情况下LinkText无法精确匹配对应的元素。对于这种情况，可以使用By.PartialLinkText进行部分文字的匹配。

被测HTML代码如下：

```
<html>
   <body>
      <div>
         <a href="https://msdn.microsoft.com/en-us/default.aspx">
            <strong>
               <i>Go</i>
            </strong>
            <i>
               -
               <i>Microsoft</i>
               <i>Development</i>
               <i>Network</i>
            </i>
         </a>
      </div>
   </body>
</html>
```

测试代码如下：

```
IWebElement element = driver.FindElement(By.PartialLinkText("Development"));
```

10.5.7　基于CssSelector定位

在前端开发中，CSS被广泛应用于页面元素的选择以及风格描述方面。WebDriver可以通过By.CssSelector以CSS选择器的语法进行元素查找。WebDriver与CSS 3.0标准兼容，支持CSS中的名字空间和伪类，可以对使用id或Name无法定位的较复杂元素进行定位。从最佳实践的经验来看，测试人员在用CSS选择器的时候，可以充分参考开发人员写的CSS文档以获得更好的匹配效果。

由于CSS选择器对页面结构的变化比较敏感，比较适合相对稳定的页面测试。以下示

例代码通过CSS选择器查找到了div2下的checkbox对象。

被测HTML代码如下：

```
<html>
   <body>
      <div id="div1">
         <input type="checkbox">Checkbox 1
      </div>
      <div id="div2">
         <input type="checkbox">Checkbox 2
      </div>
   </body>
</html>
```

测试代码如下：

```
IWebElement element = driver.FindElement(By.CssSelector("#div2 > input[type=
'checkbox']"));
```

10.5.8 基于XPath定位

XPath[①]全称是XML Path Language，是在XML文档内查找信息的语言，可以在整个文档树内通过元素和属性进行导航。XPath于1999年成为W3C标准，被设计供XSLT、XPointer以及其他XML解析软件使用，所以理解XPath对XML相关的高级应用也很有帮助。因为HTML是基于XML（XHTML）的实现，Selenium WebDriver也可以通过XPath表达式对页面内的元素进行定位。XPath作为一个强大的定位利器，大大拓展了id、Name等其他定位器的局限性，提供了丰富而灵活的导航能力。

XPath使用路径表达式来选取文档中的节点或节点集合，这些路径表达式与常规的文件系统表达式非常相似。另一方面，由于XPath使用灵活，相同的节点可以用不同的表达式定位，如何找到一个健壮的表达式对初学者有一定的学习难度。本节将基于以下被测

① W3C. XML Path Language[OL]. [2016]. https://www.w3.org/TR/xpath/.

HTML代码演示XPath的常用方法和最佳实践。

被测HTML代码如下：

```
<html>
   <body>
      <div id="div1">
         <div id="div2">
            <div id="div3">
               <input type="text" value="email" id="input1">
               <input type="text" value="phone" id="input2">
               <button id="button1">Click</button>
            </div>
         </div>
      </div>
      <div>
         <div>
            <a href="www.bing.com">Go to Bing</a>
            <input type="text">
            <input type="text">
            <input type="text">
         </div>
      </div>
   </body>
</html>
```

1. XPath辅助工具

XPath对于初学者有一定的理解难度，建议首先结合XPath辅助工具逐步练习上手。本节将介绍Firefox的两个XPath插件FirePath和WebDriver Element Locator。由于它们是Firefox的插件，安装步骤与Selenium IDE类似，本书不再赘述。

FirePath是一个基于FireBug的插件，它可以对所测节点提供XPath建议，以及对输入的XPath进行可视化验证。测试人员可以方便地通过FirePath演练XPath语法，选择最优表

达式。

示例步骤如下：

（1）启动FireFox浏览器，访问被测网页。

（2）右击页面上的Click按钮，选择Inspect in FirePath选项。

（3）如图10-8所示，在展开的FirePath标签页中，选中的按钮分别在页面和代码区内被高亮显示，同时FirePath提供了//*[@id='button1']作为建议的XPath表达式。

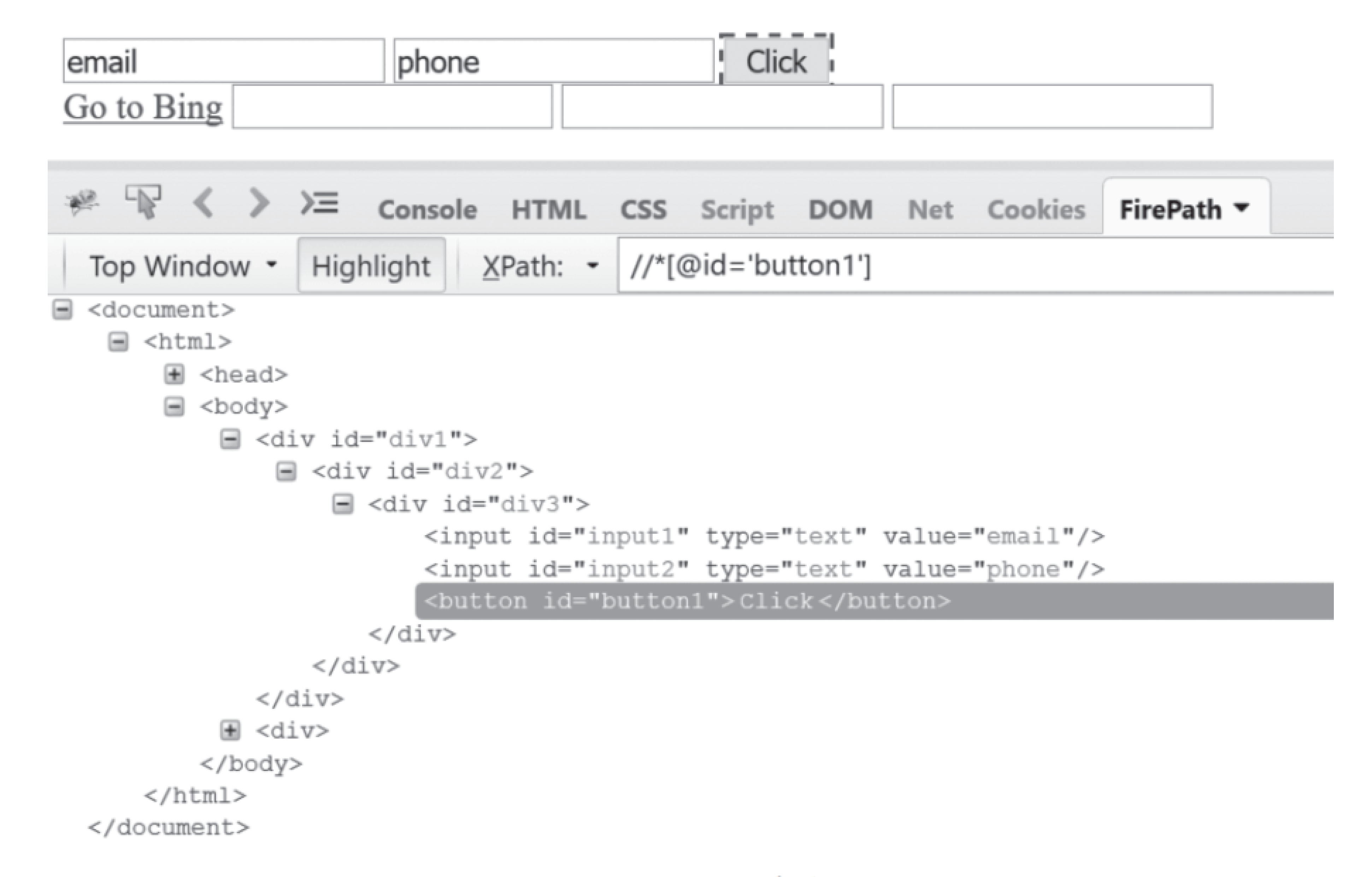


图10-8　FirePath标签页

（4）单击XPath编辑框，把其修改为//*[@id='input1']，如图10-9所示。按Enter键，则id为“input1”的input元素被高亮选中。

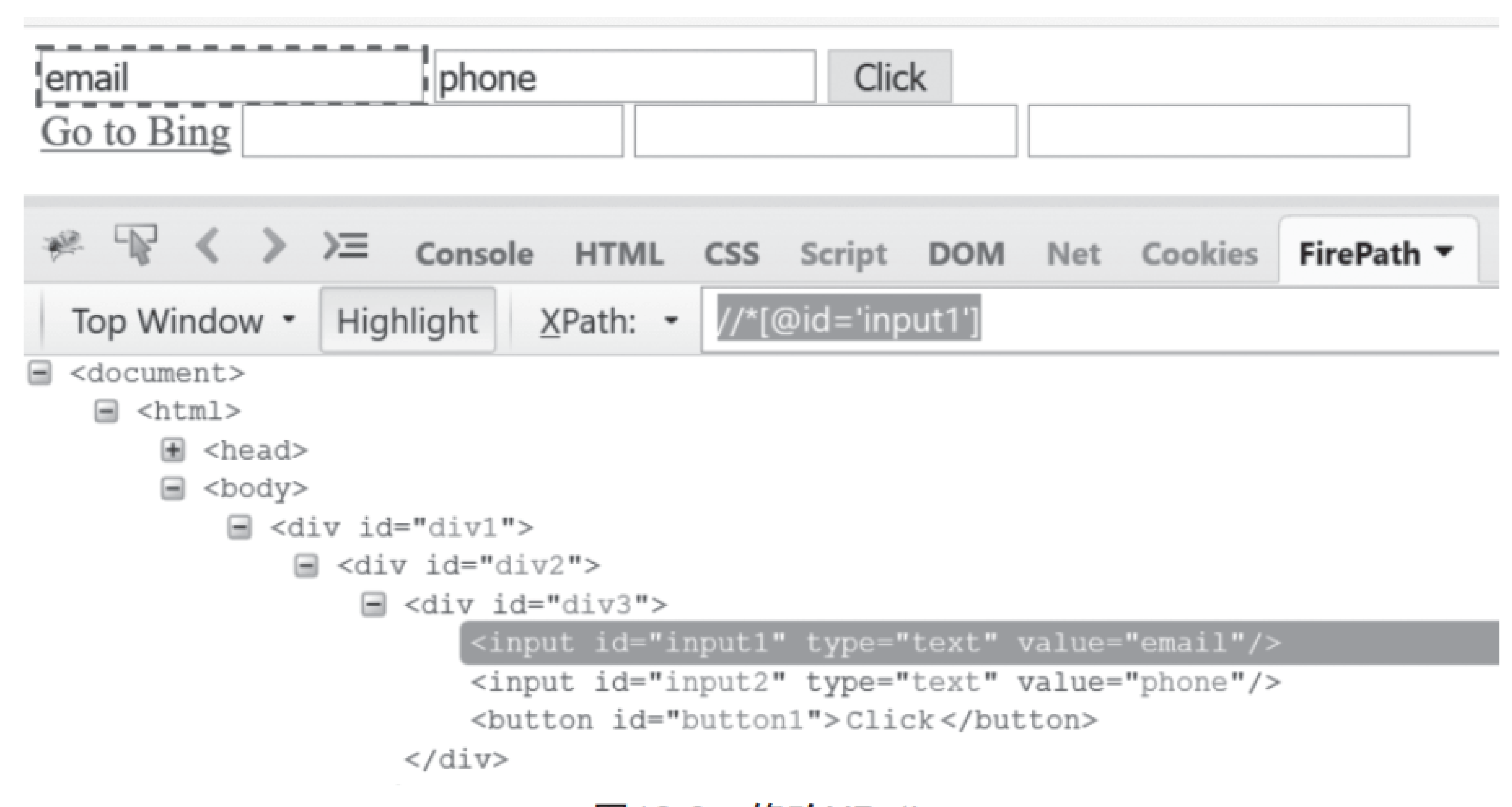


图10-9　修改XPath

WebDriver Element Locator是另一个FireFox插件，对元素定位可以提供多个XPath表达式建议，方便测试人员选择合适的定位方式，建议与FirePath结合使用。

示例步骤如下：

（1）启动FireFox浏览器，访问被测网页。

（2）右击页面上的Click按钮，选择XPaths…菜单项，如图10-10所示。

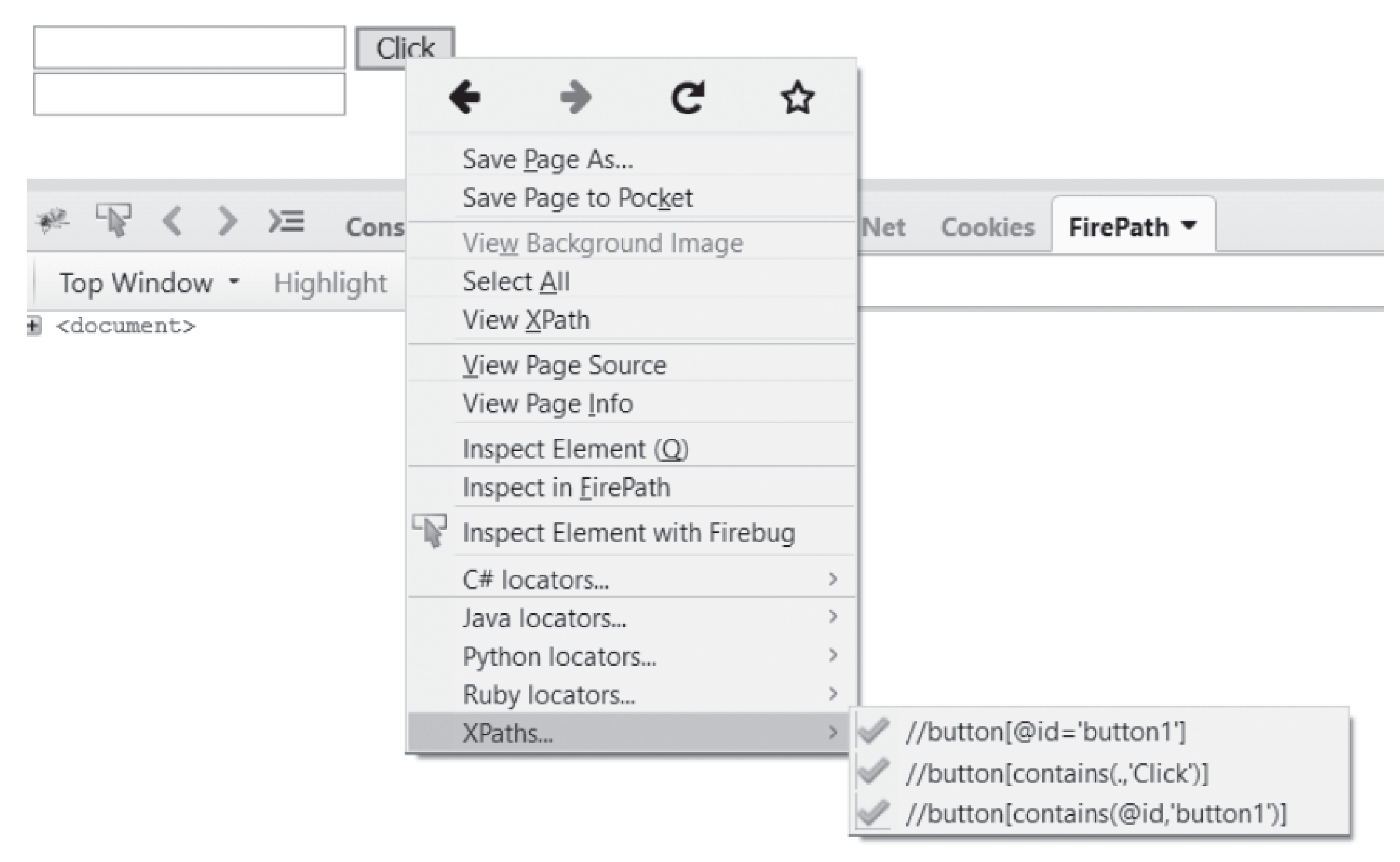


图10-10　使用WebDriver Element Locator

（3）WebDriver Element Locator 分别建议了3种方式对该按钮进行定位，选择任何一种，对应的表达式即可被复制到剪贴板内。

2. XPath定位技巧

1）通过绝对路径定位

使用了绝对路径的XPath表达式从最外层的HTML节点开始逐层根据页面结构进行查找，每层节点直接通过/进行分割。/代表从当前节点查找子节点。绝对路径的优点是效率高，但由于绝对路径完整反映了被找节点与根节点之间的关系，导致它的定位路径非常脆弱，任何文档上的结构变化都可能导致其失效，在测试过程中应该尽量避免使用绝对路径定位元素。以下代码遍历了从html到button元素的路径，通过绝对路径进行定位。

```
IWebElement element = driver.FindElement(By.XPath("html/body/div[1]/div/div/button"));
```

2）通过相对路径定位

由于绝对路径存在明显的定位不稳定性，相对路径使用得更为广泛。相对路径通过符号//从根节点或当前节点寻找匹配的子节点或孙子节点，而无需关心中间的具体准确路径。

以下代码通过相对路径定位button元素，无需指定中间的div。

```
IWebElement element = driver.FindElement(By.XPath("//button"));
```

3）通过元素属性定位

被测网页内的元素通常包含各种属性，把相对路径与这些属性结合起来往往可以唯一地标识一个元素，这种方法被广泛应用在Selenium测试中。在以下代码中，*代表任何类型的元素，@id表明匹配元素的id。

```
IWebElement element = driver.FindElement(By.XPath("//*[@id='button1']"));
```

如果将表达式改为//button[@id='button1']可以获得相同的效果，表明定位id为button1的元素。

4）通过文本匹配定位

HTML页面里最丰富的内容是文本，通过文本匹配对应的元素是Selenium测试中经常使用的匹配方式。特别是当其他属性无法唯一匹配一个元素的时候，优先推荐文本匹配。以下示例代码通过text()进行文本的精确匹配，从而定位到对应的元素。

```
IWebElement element = driver.FindElement(By.XPath("//a[text() = 'Go to Bing']"));
```

5）通过文本模糊匹配定位

针对页面中某些文本存在不确定性的情况，推荐使用模糊匹配而不是精确匹配，这样可以避免因为部分文本发生变化而导致测试失败的情况。在以下示例代码中，contains表示只需要部分匹配超链接的地址即可定位成功。

```
IWebElement element = driver.FindElement(By.XPath("//a[contains(@href, 'bing')]"));
```

6）通过多条件匹配定位

XPath不仅可以通过一个条件定位节点，还可以使用and和or逻辑操作符对多组条件进行合并定位。以下代码通过and操作，匹配同时满足value为email和type为text的元素，最终符合条件的input元素被成功定位，如图10-11所示。

```
IWebElement element = driver.FindElement(By.XPath("//*[@value='email' and @type=
'text']"));
```

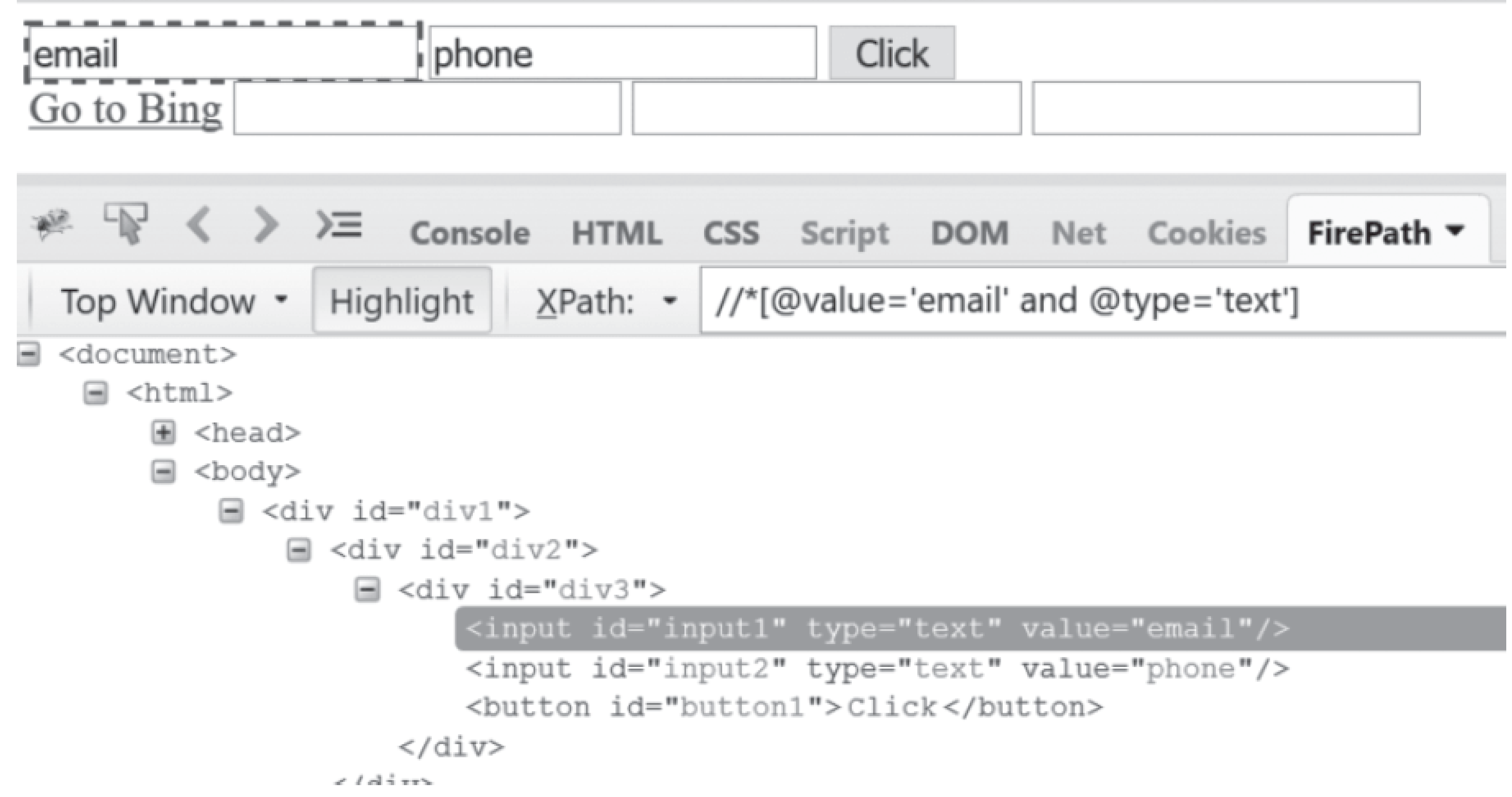


图10-11　使用and定位

以下示例代码通过关键字or进行定位，如图10-12所示，value为email和phone的两个元素被成功定位。基于or关键字的定位大多会找到多个元素。

```
var elements = driver.FindElements(By.XPath("//*[@value='email' or @value='phone']"));
```



图10-12　使用or定位

另外，也可以使用|合并多个XPath表达式。如图10-13所示，以下代码同时定位到了匹配的input元素和a元素。这种方法对于在页面内查找没有稳定位置，也没有特别关系的一批元素很有帮助。

```
var elements = driver.FindElements(By.XPath("//*[@value='email' or @value='phone'] | //a"));
```



图10-13　多表达式合并定位

7）通过索引定位

页面经常有使用多个同类型节点表示列表的情况，这些元素有完全相同的属性，无法通过id或name进行区别。针对这样的情况，可以通过索引根据元素顺序定位。以下示例代码先找到页面内的第2个div，然后依序找到里面的第2个input。

```
IWebElement element = driver.FindElement(By.XPath("//div[2]/div/input[2]"));
```

8）通过相对位置关系定位

HTML页面内有能通过id、name定位的元素，也有难以定位的元素，通过容易定位的元素找到其他元素是有效测试的必要条件。XPath提供了丰富的基于相对位置定位的关键字，常用的如表10-3所示。

表10-3　常用XPath相对位置定位关键字

关键字名称	作　　用	示 例 代 码	示 例 说 明
parent	查找当前节点的父节点	//*[@id='button1']/parent::*/input	定位到button的父节点后继续寻找input节点
ancestor	查找当前节点的所有上层节点	//*[@id='div2']/ancestor::div	在div2节点的上层节点里找到所有div类型的节点
descendant	查找当前节点的所有下层节点	//*[@id='div1']/descendant::button	在div1节点的下层节点里找到所有button类型的节点

第11章 基于WebDriver的Protractor测试框架

Protractor是基于Selenium WebDriver的自动化测试框架，编程语言为JavaScript。Protractor不仅具有所有WebDriver的优点，对AngularJS应用的测试还提供了原生支持。

本章将介绍：

- WebDriver的JavaScript绑定
- 搭建Protractor测试环境
- 选择JavaScript测试框架
- 定位页面元素
- 异步流程控制
- 页面交互
- Protractor的等待机制
- 测试非AngularJS程序

11.1 WebDriver的JavaScript绑定

正如第10章所述，WebDriver支持多种编程语言，例如C#和Java等。作为Web世界的主要语言，JavaScript也理所当然地进入了WebDriver的支持列表，于是就迎来了WebDriverJs[①]。读者可能会奇怪，C#和Java作为Selenium世界的中坚力量，已经非常稳

① SeleniumHQ. WebDriverJs[OL]. [2016]. https://github.com/SeleniumHQ/selenium/wiki/WebDriverJs.

定，也被证明是良好高效的Selenium自动化测试语言，为什么还要考虑JavaScript呢？

（1）正如第1章所言，目前的前端开发正在加速转型，JavaScript已经是编程语言里特别是Web开发中越来越重要的一份子。

（2）当前软件开发的趋势是DevOps一体化，过去开发测试以不同的组织结构分开进行的时代已经一去不复返。在敏捷开发理论的指导下，技术人员的职位界限将进一步模糊，一位工程师往往同时承担开发、测试、集成和运维的工作。也就是说用与开发相同的语言进行自动化测试已经成为了DevOps的基石之一。

（3）基于Node.js的开发技术已经非常成熟并得到了广泛的使用，这意味着越来越多掌握了JavaScript的技术人员希望用他们最熟悉的JavaScript编程语言进行自动化测试的开发工作。

11.1.1 WebDriverJs与Protractor

作为WebDriver大家庭的一员，WebDriverJs是Selenium官方对JavaScript绑定的实现。WebDriverJs既具有与C#、Java等其他WebDriver编程语言一致的功能，又兼顾了JavaScript平台，可以运行在Node.js环境内。WebDriverJs对各大主流浏览器都有着良好支持，包括Chrome、Internet Explorer、Edge、Firefox、Opera、Safari和PhantomJS。

为了使用WebDriverJs，需要在项目根目录执行以下命令安装WebDriverJs：

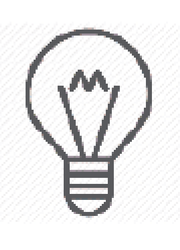
```
npm install selenium-webdriver
```

安装完WebDriverJs后，编写以下JavaScript代码，即可通过Builder API创建出Chrome的驱动对象，并打开必应主页。

```
var webdriver = require('selenuim-webdriver');
var driver = new webdriver.Builder().
   withCapabilities(webdriver.Capabilities.chrome()).
   build();
driver.get('https://www.bing.com');
```

在WebDriverJs的基础上，Google研发了一款自动化测试框架，这就是Protractor。由于它底层是对selenium-webdriver的封装，所以拥有所有WebDriver的优点和功能，而且添

加了更多的新功能。为了提高项目开发效率与代码的健壮性，Protractor选择了TypeScript①作为开发语言，它是JavaScript的一个超集，并扩展了语言特效，包括实现类、接口、模块、类型检查等。



不要将Protractor与测试框架例如Jasmine等混淆。Protractor是对selenium-webdriver的封装，并不是一个独立的测试工具，它需要与Jasmine等测试框架结合使用才能实现自动化测试。

11.1.2　Protractor特点概述

说到Protractor，就不得不提AngularJS。在Protractor之前，对AngularJS的页面进行自动化测试是有相当难度的。我们知道，AngularJS的页面渲染是通过在$digest循环中检查并更新数据完成的，这是AngularJS独有的特点。遗憾的是，WebDriverJs并不是专门为AngularJS研发的，它并不理解这个循环过程。所以，如果直接使用WebDriver对AngularJS页面进行测试，需要大量的显式等待代码确保$digest循环结束和页面刷新完成，这样的代码维护难度高，可读性差。

另一方面，AngularJS是一个可以对HTML进行扩展的语言，它不仅提供了大量原生的Directive，更让开发者有机会开发自己的Directive。这些都是宝贵的页面开发知识，对自动化测试的定位很有帮助。那么，在当前DevOps的指导下，如何让技术人员充分基于开发中使用的技术在自动化测试中写出健壮的测试代码呢？

针对以上需求，AngualrJS的开发团队早期通过Karma和ngScenario②进行端到端的自动化测试。ngScenario提供了一套类似于Selenium的接口来驱动浏览器，但是后续遇到了越来越多功能上的局限性，基于ngScenario的思路，AngularJS团队决定主要用Karma来驱动单元测试，而基于ngScenario的思路另外开发了Protractor来专门进行自动化测试。

以下是Protractor的主要特点：

- 自动执行等待，无需专门的代码进行页面同步。
- 针对AngularJS提供专有元素定位方式。
- 同时支持AngularJS和非AngularJS的应用。
- 使用相同的JavaScript语言进行开发、单元测试和自动化测试。

① TypeScript. TypeScript[OL]. [2016]. http://www.typescriptlang.org/.

② Karma-ng-scenario. Karma-ng-scenario[OL]. [2016]. https://www.npmjs.com/package/karma-ng-scenario.

- 可以搭配多种BDD测试框架，包括Jasmine、Mocha和Cucumber等。
- 支持多种浏览器，包括IE、Chrome、Safari、FireFox和PhantomJS等。
- 使用方便，对于已经在用Node.js的团队非常容易上手。
- 同时支持本地和远程测试。
- 配置灵活，方便持续集成。

Protractor有这么多优点，是不是意味着自动化测试一定要首选JavaScript和Protractor呢？答案是否定的。Selenium WebDriver是一个百花齐放的开放社区，这里不仅有最成熟的C#和Java，也有近几年异军突起的Python，不同的语言都能找到自己的适用范围。另外，在进行框架选型的时候，也要综合考虑本公司的技术积累，一般建议从较熟悉的编程语言入手。

11.1.3 Protractor的兼容性

在作者编写本书的时候，Protractor最新的版本是Protractor 4，它与Node.js 4.0及以上版本兼容。如果必须使用Node.js 0.12，则需要用Protractor 2。

Protractor支持1.1.4之后的AngularJS应用。如果需要测试版本更早的AngularJS应用，可以考虑WebDriverJS或者ngScenario。

11.2 搭建Protractor测试环境

在详细介绍Protractor的接口和配置之前，为了帮助读者快速入门，本节将基于一个简单的例子来演示如何搭建Protractor的测试环境以及编写第一个Protractor脚本。由于Protractor是基于Node.js的，请读者在进行以下步骤前确保Node.js已经正确安装，具体步骤可以参考本书第2章。

11.2.1 安装Protractor编辑器扩展

Protractor自动化测试的核心工作是编写测试用例，也就是编写JavaScript代码，所以理论上任何编辑器都可以胜任，读者在实际工作中可以根据喜好或习惯选择自己所用的编

辑器。本书所有的Protractor测试用例都使用Visual Studio Code进行编辑，由于它对Node.js提供了原生支持，可以无缝支持Protractor的编辑和调试。另外，读者也可以在Visual Studio Code里安装Protractor的扩展插件，它提供了丰富的Protractor代码智能提示，可提高脚本的开发效率。其具体安装步骤如下：

（1）启动Visual Studio Code，单击Extensions项，如图11-1所示。

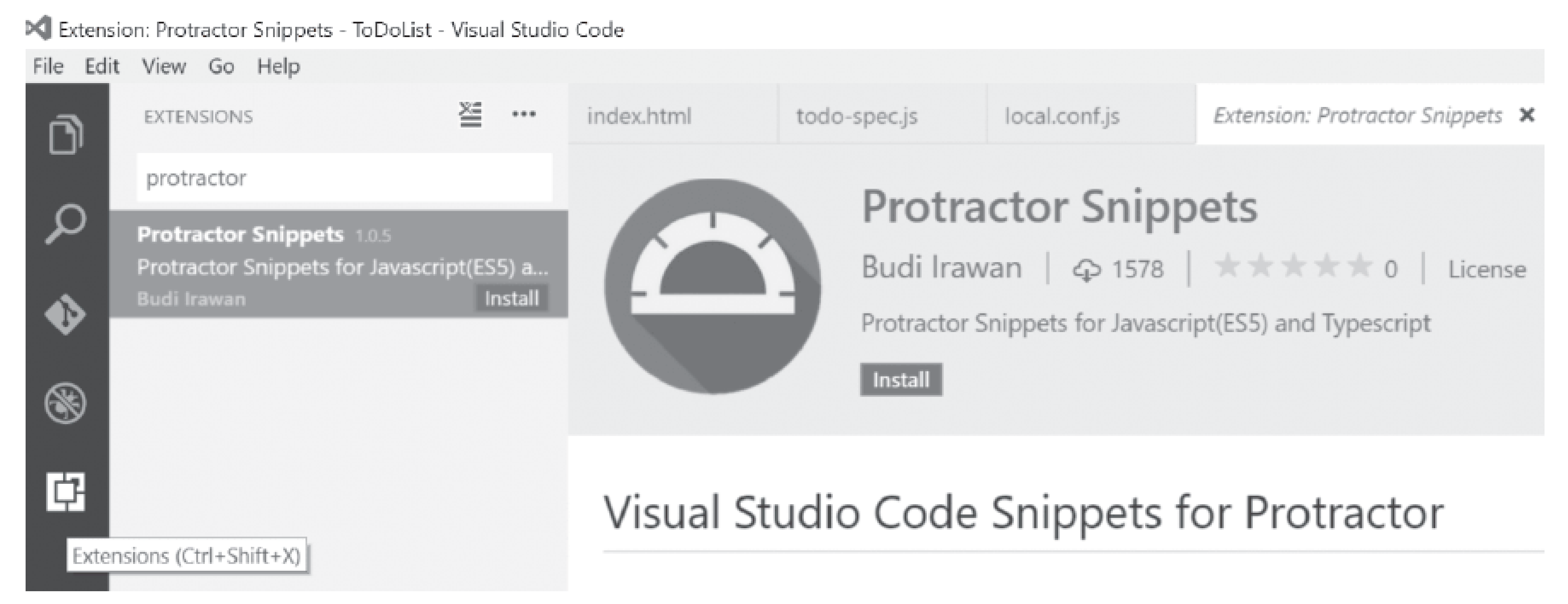


图11-1　安装Protractor插件

（2）在搜索框键入Protractor后找到Protractor Snippets项，单击Install按钮。

11.2.2　准备AngularJS被测网站

首先创建本地文件夹ToDoList，并添加被测页面的相关文件index.html和todo.js。两者内容分别如下：

index.html

```
<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="todo.css">
  </head>
  <body  ng-app="todoApp">
    <h2>Todo</h2>
    <div ng-controller="TodoListController as todoList">
      <span>{{todoList.remaining()}} of {{todoList.todos.length}} remaining</span>
      [ <a href="" ng-click="todoList.archive()">archive</a> ]
```

```
    <ul class="unstyled">
      <li ng-repeat="todo in todoList.todos">
        <label class="checkbox">
          <input type="checkbox" ng-model="todo.done">
          <span class="done-{{todo.done}}">{{todo.text}}</span>
        </label>
      </li>
    </ul>
    <form ng-submit="todoList.addTodo()">
      <input type="text" ng-model="todoList.todoText"  size="30"
             placeholder="add new todo here">
      <input class="btn-primary" type="submit" value="add">
    </form>
  </div>
      <script src="angular.min.js"></script>
      <script src="todo.js"></script>
  </body>
</html>
```

todo.js

```
angular.module('todoApp', [])
  .controller('TodoListController', function() {
    var todoList = this;
    todoList.todos = [
      {text:'learn angular', done:true},
      {text:'build an angular app', done:false}];
    todoList.addTodo = function() {
      todoList.todos.push({text:todoList.todoText, done:false});
      todoList.todoText = '';
    };
```

```
    todoList.remaining = function() {
      var count = 0;
      angular.forEach(todoList.todos, function(todo) {
        count += todo.done ? 0 : 1;
      });
      return count;
    };
    todoList.archive = function() {
      var oldTodos = todoList.todos;
      todoList.todos = [];
      angular.forEach(oldTodos, function(todo) {
        if (!todo.done) todoList.todos.push(todo);
      });
    };
  });
Todo.js
.done-true {
  text-decoration: line-through;
  color: grey;
}
```

启动命令控制台，执行命令http-server启动本地Web服务器（如果读者尚未安装http-server，请参考4.7.2节的安装步骤）。被测页面可以通过8080端口进行访问，如图11-2所示。这是一个Todo列表应用，用户可以添加新的任务，或者标注任务为完成状态。

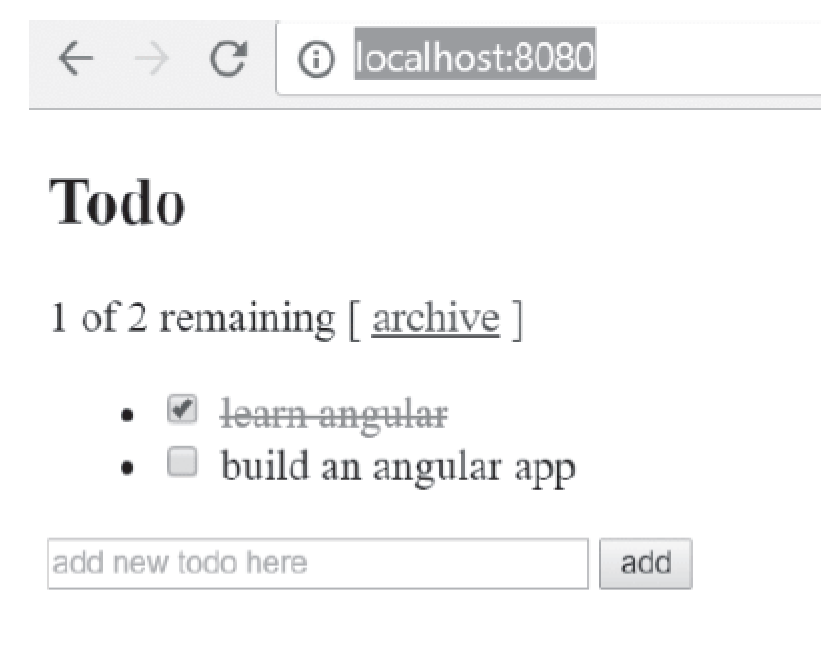


图11-2　被测程序

11.2.3 全局安装Protractor与浏览器驱动

启动命令控制台，通过以下命令把Protractor安装到全局缓存内，这会让Protractor 命令行接口可以作用到全局。

```
npm install -g protractor
```

安装完成后可以键入以下命令检查安装是否成功，成功的话会返回Protractor当前的版本。

```
protractor -version
```

Protractor既可以通过文件配置测试条件，也接受参数。执行以下命令可以得到所有参数的帮助，如图11-3所示。

```
protractor -help
```

```
C:\>protractor -help
**you must either specify a configuration file or at least 3 options. See below for the options:

Usage: protractor [configFile] [options]
configFile defaults to protractor.conf.js
The [options] object will override values from the config file.
See the reference config for a full list of options.

Options:
  --help                              Print Protractor help menu
  --version                           Print Protractor version
  --browser, --capabilities.browserName  Browsername, e.g. chrome or firefox
  --seleniumAddress                   A running selenium address to use
  --seleniumSessionId                 Attaching an existing session id
  --seleniumServerJar                 Location of the standalone selenium jar file
  --seleniumPort                      Optional port for the selenium standalone server
  --baseUrl                           URL to prepend to all relative paths
  --rootElement                       Element housing ng-app, if not html or body
  --specs                             Comma-separated list of files to test
  --exclude                           Comma-separated list of files to exclude
  --verbose                           Print full spec names
  --stackTrace                        Print stack trace on error
  --params                            Param object to be passed to the tests
  --framework                         Test framework to use: jasmine, mocha, or custom
  --resultJsonOutputFile              Path to save JSON test result
  --troubleshoot                      Turn on troubleshooting output
  --elementExplorer                   Interactively test Protractor commands
  --debuggerServerPort                Start a debugger server at specified port instead of repl
```

图11-3 Protractor命令行参数

全局安装完Protractor后，执行以下命令，将Chrome等浏览器驱动下载到全局目录中。

```
webdriver-manager update
```

11.2.4　本地安装Protractor与浏览器驱动

在命令控制台依次执行以下命令，初始化package.json文件，将Protractor安装到本地，并下载浏览器驱动。与上一节将Protractor安装到全局不同，本地安装只会作用到当前项目文件夹内，多用于构建工具的集成。

```
npm init
npm install protractor --save-dev
node .\node_modules\protractor\node_modules\webdriver-manager update
```

11.2.5　编写测试代码

用Visual Studio Code创建新文件todo-spec.js，代码如下：

```
describe('todo list', function() {
  it('should add a todo', function() {
    browser.get('/');
    element(by.model('todoList.todoText')).sendKeys('first script');
    element(by.css('[value="add"]')).click();
    var todoList = element.all(by.repeater('todo in todoList.todos'));
    expect(todoList.count()).toEqual(3);
    expect(todoList.get(2).getText()).toEqual('first script');
    todoList.get(2).element(by.css('input')).click();
    var completedAmount = element.all(by.css('.done-true'));
    expect(completedAmount.count()).toEqual(2);
  });
});
```

以上测试用例依旧是由大家已经熟悉的Jasmine单元测试框架来组织，具体用法请参考第4章。

11.2.6 编写配置文件

在ToDoList文件夹内创建配置文件local.conf.js，Protractor将通过该配置文件决定测试使用的环境，代码如下：

```
exports.config = {
  directConnect: true,
  specs: ['todo-spec.js'],
  baseUrl: 'http://localhost:8080',
  framework: 'jasmine2'
};
```

配置说明：

- baseUrl代表了被测网站的地址，http://localhost:8080表示将要测试的ToDoList网站被启动在了本地的8080端口。如果读者想直接对AngularJS的官方网址进行测试，请将其修改为https://angularjs.org/。
- directConnect表示Protractor直接操作驱动进行浏览器操作，默认使用Chrome作为测试浏览器。
- specs是一个数组，定义测试用例的范围。在本例中，只有一个测试用例todo-spec.js。
- framework指定使用什么测试框架组织和驱动测试用例。本例使用的是Jasmine，也是Protractor的默认测试框架。

11.2.7 运行测试用例

启动另一个命令控制台，并设置当前路径为ToDoList，执行以下命令运行测试脚本。Chrome是Protractor测试的默认浏览器，在本例中Chrome将被Protractor启动并执行测试用例。如图11-4所示，第一个脚本顺利通过测试。

```
protractor local.conf.js
```

```
C:\ToDoList>protractor local.conf.js
[16:45:49] I/direct - Using ChromeDriver directly...
[16:45:49] I/launcher - Running 1 instances of WebDriver
Started
.


1 spec, 0 failures
Finished in 1.127 seconds

[16:45:52] I/launcher - 0 instance(s) of WebDriver still running
[16:45:52] I/launcher - chrome #01 passed
```

图11-4　运行Protractor

本测试用例todo-spec.js使用到了browser对象，这是Protractor对WebDriver实例的包装，作为Protractor的核心对象，可以通过browser对象实现页面导航以及页面信息的提取。

在本例中，还可以看到by.model和by.repeater这两种全新的定位方式。读者可能还记得第10章介绍的传统的8种定位方式中并没有包含这两种，这是Protractor为AngularJS专门带来的定位方式，本书将在11.4节中做详细介绍。

11.2.8　调试

Visual Studio Code原生支持Node.js，可以方便地调试JavaScript、TypeScript以及其他用于生成JavaScript的编程语言。以下为调试Protractor的配置步骤。

（1）在Visual Studio Code左侧工具栏单击Debug按钮，如图11-5所示。注意，当前显示No Configurations，表示当前还没有配置文件定义调试选项。

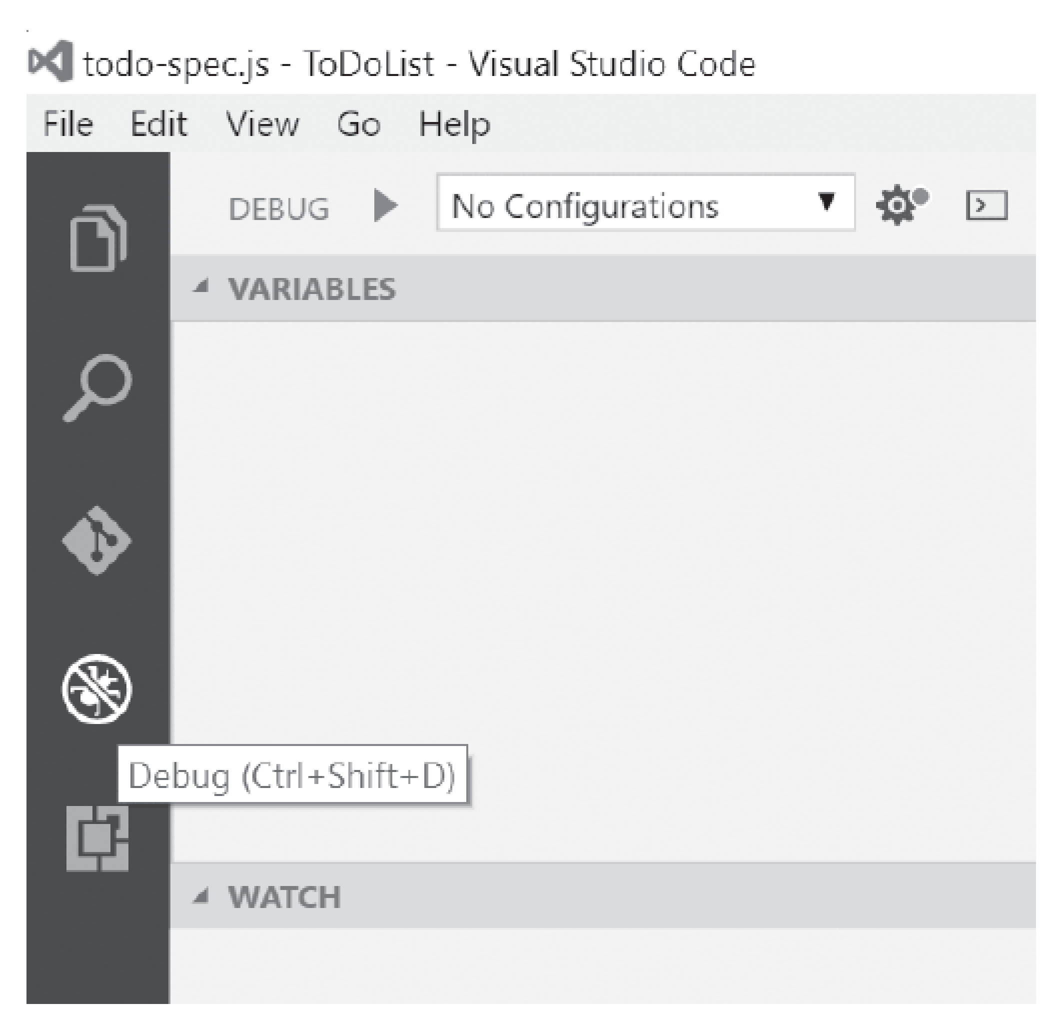


图11-5　开始配置调试环境

（2）单击“Configure or Fix 'launch.js'”齿轮按钮，如图11-6所示。

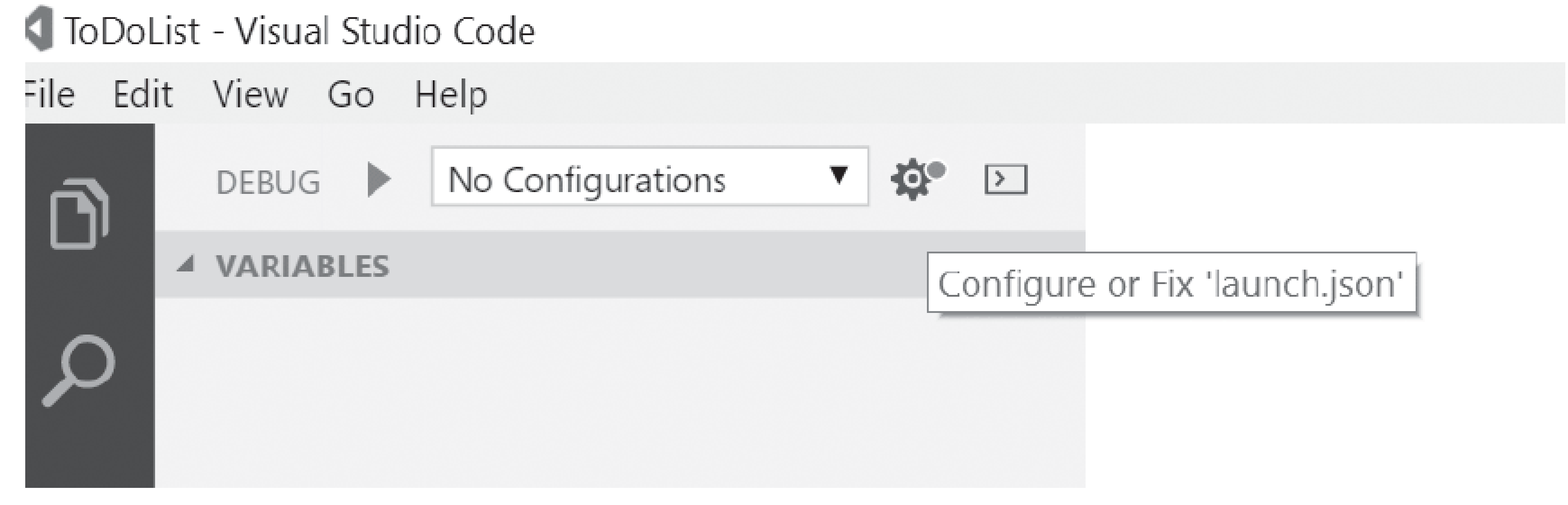


图11-6 配置调试启动项

（3）选择Node.js作为调试环境，如图11-7所示。

Select Environment
Node.js
VSCode Extension Development

图11-7 选择Node.js

（4）配置文件launch.json被自动创建到.vscode文件夹内，该文件包含了用于调试Node.js应用的配置项。找到name为Launch的配置块，修改代码如下：

```
{
            "name": "Launch",
            "type": "node",
            "request": "launch",
            "program": "${workspaceRoot}/node_modules/protractor/bin/protractor",
            "stopOnEntry": false,
            "args": ["${workspaceRoot}/local.conf.js"],
            "cwd": "${workspaceRoot}",
            "preLaunchTask": null,
            "runtimeExecutable": null,
            "runtimeArgs": [
               "--nolazy"
            ],
            "env": {
               "NODE_ENV": "development"
            },
```

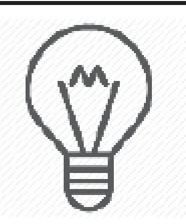
```
            "externalConsole": false,

            "sourceMaps": false,

            "outDir": null

}
```

其中，program字段表示用protractor驱动测试，args字段表示把local.conf.js作为参数传给protractor。



launch.json中的Launch仅仅是该配置项的名字，launch.json支持任意名字的配置项，从而能够以不同的参数配置调试①。

（5）打开todo-spec.js，按F9键设置断点，如图11-8所示。单击DEBUG按钮后启动调试。

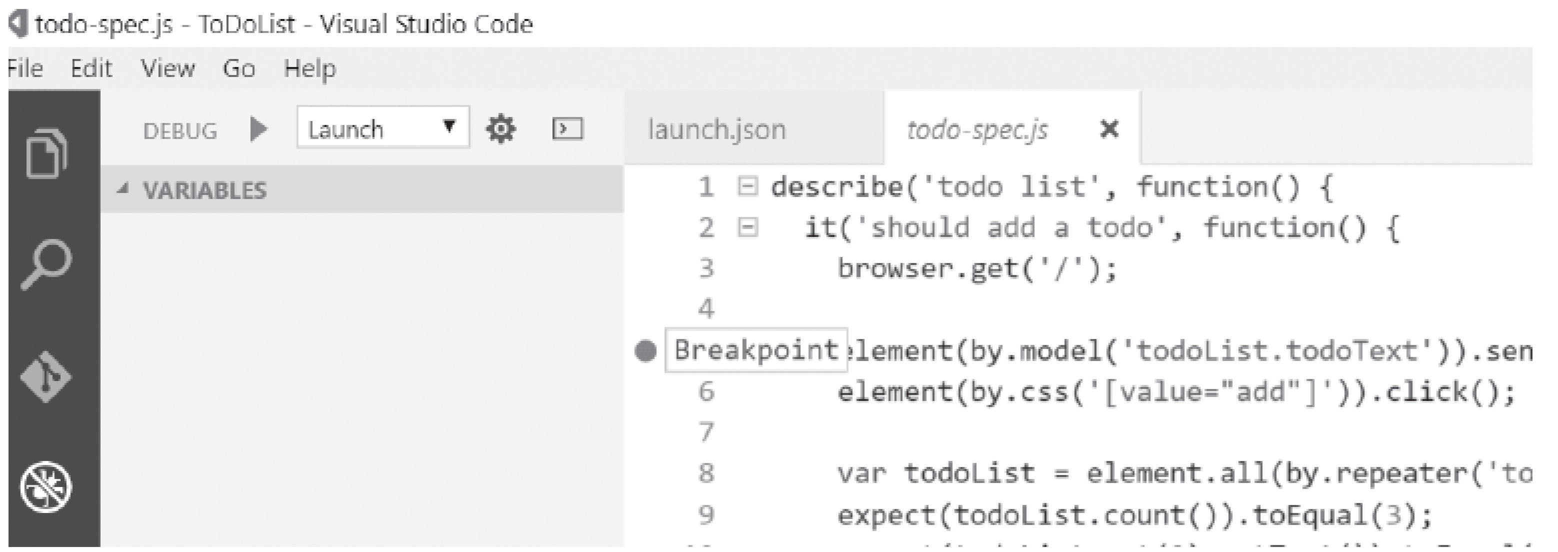


图11-8　设置断点

（6）如图11-9所示，Visual Studio Code在断点处会自动中断执行，可以在VARIABLES和WATCH窗口观察变量的值。



图11-9　调试

① Wu Shuai. Debug protractor script in Visual Studio Code[OL]. 2016. https://blogs.msdn.microsoft.com/wushuai/2016/08/24/debug-protractor-script-in-visual-studio-code/.

11.3 选择JavaScript测试框架

正如在C#测试脚本中通过NUnit组织测试用例，Protractor同样需要JavaScript单元测试框架的支持。Protractor利用适配器与测试框架集成，这样的设计保证其既支持主流的BDD框架例如Jasmine、Mocha等，也能够支持其他测试框架，读者在选型的时候可以根据本公司的知识储备加以选择。

如图11-10所示，由于Protractor已经自带了Jasmine和Mocha的适配器，无需额外下载即可直接使用Jasmine和Mocha；对于其他测试框架，读者可以到开源社区下载或者自行实现适配器。

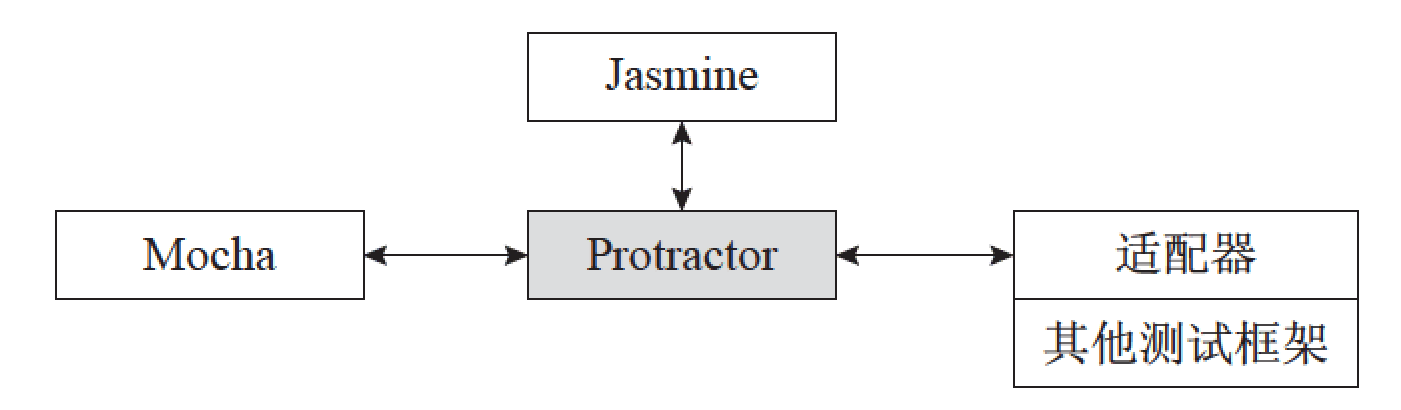


图11-10　Protractor测试框架

Jasmine和Mocha的适配器源码可以从protractor/built/framework/jasmine.js和protractor/built/frameworks/mocha.js处获得。

Jasmine 2.x是Protractor的默认测试框架，会作为Protractor的依赖包被一起下载到计算机。考虑到本书单元测试篇是基于Jasmine构建，而使用相同的单元测试框架组织测试用例可以大大降低开发人员的学习成本，因此本书所有Protractor测试用例均将基于Jasmine来组织。

11.3.1 配置JavaScript测试框架

JavaScript测试框架往往提供了多种配置参数用于控制测试行为或者报表格式，那么如何在Protractor里设置这些参数呢？

以Jasmine为例，它对外提供了配置选项，如表11-1所示。

表11-1　Jasmine配置参数

名　　称	类型	描　　述
showColors	boolean	是否在控制台终端打印颜色，默认为true
defaultTimeoutInterval	number	在标注一个测试用例失败前的等待时间，单位是毫秒，默认为30000毫秒

（续表）

名　　称	类型	描　　述
print	function	回调函数，用于打印测试结果。如果为空，Jasmine会添加自己的实现
grep	string	正则表达式，只执行符合匹配的测试用例，可以为空
invertGrep	boolean	是否反转grep正则表达式
jasmineCorePath	string	指定Jasmine-Core的安装路径，可以为空，默认指向Jasmine-npm内的依赖包

可以在Protractor的配置文件中添加jasmineNodeOpts，配置Jasmine相关的选项。代码如下：

```
exports.config = {
  directConnect:true,
  specs: ['todo-spec.js'],
  baseUrl: 'http://localhost:8080',
  framework: 'jasmine2',
  jasmineNodeOpts:{
    defaultTimeoutInterval: 10000,
    showColor: false
  }
};
```

11.3.2　JavaScript测试框架的适配器

一个完整的测试流程包括解析传入到Protractor CLI的参数，基于配置选择测试框架，驱动测试用例。这些步骤主要由以下组件完成（参见图11-11）：

- 启动器：解析配置文件中浏览器选项和测试框架选项并传递给驱动器。
- 驱动器：根据配置选型选择测试框架的种类。执行测试用例并触发测试用例启动或结束的事件。
- 适配器：驱动器和测试框架的桥梁，初始化测试框架的配置选项，添加测试用例。
- 测试框架：测试用例的实际执行者。不同的测试框架需要不同的适配器与Protractor进行集成。

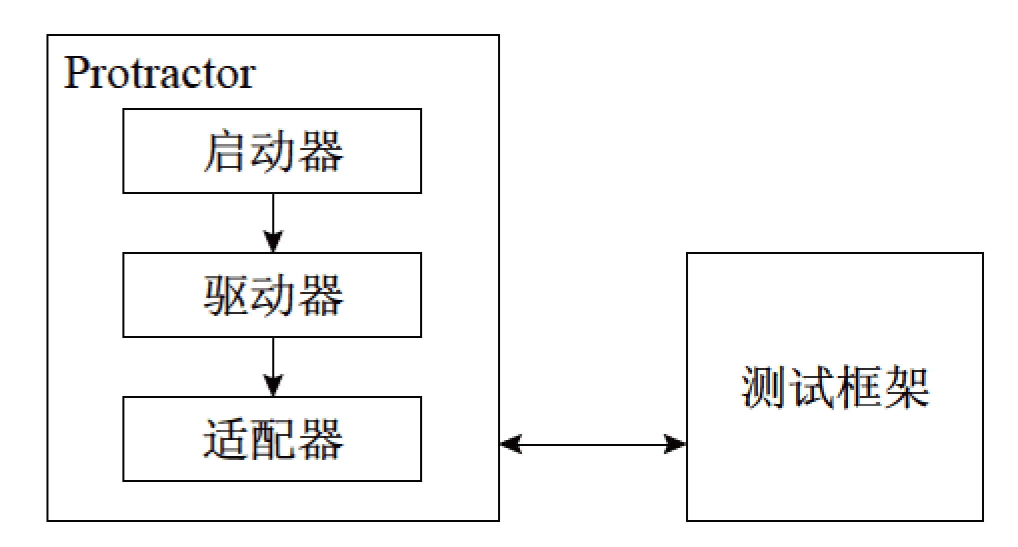


图11-11　Protractor测试组件

以下为Jasmine适配器的部分源码，供读者参考：

```
var JasmineRunner = require('jasmine');
var jrunner = new JasmineRunner();
/* global jasmine */
require('jasminewd2');
var jasmineNodeOpts = runner.getConfig().jasmineNodeOpts;
// On timeout, the flow should be reset. This will prevent webdriver tasks
// from overflowing into the next test and causing it to fail or timeout
// as well. This is done in the reporter instead of an afterEach block
// to ensure that it runs after any afterEach() blocks with webdriver tasks
// get to complete first.
var reporter = new RunnerReporter(runner);
jasmine.getEnv().addReporter(reporter);
// Filter specs to run based on jasmineNodeOpts.grep and jasmineNodeOpts.invert.
  jasmine.getEnv().specFilter = function(spec) {
    var grepMatch = !jasmineNodeOpts ||
        !jasmineNodeOpts.grep ||
        spec.getFullName().match(new RegExp(jasmineNodeOpts.grep)) != null;
    var invertGrep = !!(jasmineNodeOpts && jasmineNodeOpts.invertGrep);
    if (grepMatch == invertGrep) {
      spec.pend();
    }
    return true;
```

```
  };
  return runner.runTestPreparer().then(function() {
    return q.promise(function(resolve, reject) {
      if (jasmineNodeOpts && jasmineNodeOpts.defaultTimeoutInterval) {
        jasmine.DEFAULT_TIMEOUT_INTERVAL = jasmineNodeOpts.defaultTimeoutInterval;
      }
      var originalOnComplete = runner.getConfig().onComplete;
      jrunner.onComplete(function(passed) {
        try {
          var completed = q();
          if (originalOnComplete) {
            completed = q(originalOnComplete(passed));
          }
          completed.then(function() {
            resolve({
              failedCount: reporter.failedCount,
              specResults: reporter.testResult
            });
          });
        } catch (err) {
          reject(err);
        }
      });
      jrunner.configureDefaultReporter(jasmineNodeOpts);
      jrunner.projectBaseDir = '';
      jrunner.specDir = '';
      jrunner.addSpecFiles(specs);
      jrunner.execute();
    });
});
```

11.4 定位页面元素

Protractor作为对WebDriverJs的封装，提供了多样化的定位策略来查找页面元素，充分利用AngularJS特有的属性，使定位更实用、更方便。Protractor通过提供以下3个对象实现对浏览器的操作和元素定位：

（1）browser：代表当前浏览器的一个实例，是一个对WebDriver的包装，主要用于页面浏览以及获得页面信息。

（2）by：元素定位策略选择器，决定用什么方式定位元素。

（3）element：功能函数，结合by实现元素定位并返回定位到的元素。

为了方便编写脚本，以上3个对象在Protractor初始化过程中由protractor\built\runner.js暴露为全局变量，代码如下：

```
global.browser = browser_;
global.$ = browser_.$;
global.$$ = browser_.$$;
global.element = browser_.element;
global.by = global.By = ptor_1.protractor.By;
```

功能函数element接受一个定位策略Locator对象作为参数，返回单个ElementFinder对象。ElementFinder可以理解为一个WebElement的包装，通过它可以实现DOM元素的操作。Locator对象的创建通过全局的by对象实现。还记得第10章本书介绍的8种WebDriver原生定位方式吗？作为WebDriver的包装，Protractor仍然支持这8种定位方式，如表11-2所示。从表11-2中可以看到element接受了by指定的定位策略然后返回查找到的元素对象。

表11-2　Protractor的定位方法

定位方式	示　　例
Id	element(by.id('dog_id'));
Name	element(by.name('dog_name'));
ClassName	element(by.className('dog'));
TagName	element(by.tagName('a'));
LinkText	element(by.linkText('Protractor'));
PaitialLinkText	element(by.partialLinkText ('Protractor'));
CssSelector	element(by.css('.pet .cat'));
XPath	element(by.xpath('//ul/li/a'));

如果开发人员想返回多个元素，可以使用element.all函数。该函数接受一个Locator对象并返回ElementArrayFinder对象，可以理解为一组WebElement的集合。

除了以上原生的定位策略，Protractor还提供了新的定位策略，特别是对AngularJS的特殊属性进行了优化。

11.4.1　基于binding定位

ng-bind是AngularJS里进行数据绑定的方法，基于by.binding，Protractor可以通过HTML页面内使用的绑定字符串定位元素。by.binding支持模糊匹配，通过部分字符串进行匹配。

被测HTML代码如下：

```
<span>{{person.name}}</span>
<span ng-bind="person.email"></span>
```

定位代码如下：

```
element(by.binding('person.name'));
element(by.binding('person.email'));
element(by.binding('name')); //partial match
```

以上代码中，element和by就是暴露在global对象上的变量，它们也都可以通过另一个全局变量protractor间接获得，例如：

```
protractor.browser.element(protractor.by.binding('person.name'));
protractor.browser.element(protractor.by.binding('person.email'));
protractor.browser.element(protractor.by.binding('name'));
```

出于代码简洁的最佳实践，推荐直接使用element和by。另外，Protractor也提供了by.exactBinding进行精确定位，避免于模糊定位返回多个元素的情况，例如：

```
element(by.exactBinding('person.name'));
```

Protractor新添加的所有定位方法都基于WebDriver实现，以下为by.exactBinding的源代码。

```
exactBinding(bindingDescriptor: string): Locator {
    return {
      findElementsOverride:
          (driver: WebDriver, using: WebElement, rootSelector: string):
              wdpromise.Promise<WebElement[]> => {
            return driver.findElements(webdriver.By.js(
                clientSideScripts.findBindings, bindingDescriptor, true, using,
                rootSelector));
          },
      toString: (): string => {
        return 'by.exactBinding("' + bindingDescriptor + '")';
      }
    };
};
```

如果读者有兴趣可以在网址https://github.com/angular/protractor/blob/4.0.9/lib/locators.ts处看到其他定位方法的源代码。

11.4.2 基于model定位

使用by.model通过检查ng-model表达式定位元素。

被测HTML代码如下：

```
<span ng-model="person.email"></span>
```

定位代码如下：

```
element(by.model('person.name'));
```

11.4.3 基于options定位

使用by.options通过检查ng-options表达式定位元素。

被测HTML代码如下：

```
<select ng-model="color" ng-options="c for c in colors">
  <option value="0" selected="selected">red</option>
  <option value="1">green</option>
</select>
```

定位代码如下：

```
element.all(by.options('c for c in colors'));
```

11.4.4　基于buttonText定位

使用by.buttonText通过按钮文字定位元素。

被测HTML代码如下：

```
<button>Save my file</button>
```

定位代码如下：

```
element(by.buttonText('Save my file'));
```

同时，Protractor提供了by.partialButtonText进行模糊定位，代码如下：

```
element(by.partialButtonText('Save'));
```

11.4.5　基于repeater定位

ng-repeater是非常实用的AngularJS指令，广泛用于绑定列表。使用by.repeater函数，可以通过ng-repeater表达式定位元素。

被测HTML代码如下：

```
<div ng-repeat="cat in pets">
  <span>{{cat.name}}</span>
  <span>{{cat.age}}</span>
</div>
```

定位代码如下：

```
// Returns the DIV for the second cat.
var secondCat = element(by.repeater('cat in pets').row(1));
// Returns the SPAN for the first cat's name.
var firstCatName = element(by.repeater('cat in pets').
row(0).column('cat.name'));
// Returns a promise that resolves to an array of WebElements from a column
var ages = element.all(by.repeater('cat in pets').column('cat.age'));
```

11.4.6 基于js定位

Protractor提供了by.js函数，可以利用自定义JavaScript代码对被测页面元素进行定位。自定义JavaScript代码作为字符串或者回调函数传入by.js，这些代码的执行上下文在被测页面内，并不能与测试代码本身交互，即无法引用测试代码内的变量。使用by.js函数的示例代码如下。

被测HTML代码如下：

```
<span class="small">One</span>
<span class="medium">Two</span>
<span class="large">Three</span>
```

定位代码如下：

```
var wideElement = element(by.js(function() {
  var spans = document.querySelectorAll('span');
  for (var i = 0; i < spans.length; ++i) {
    if (spans[i].offsetWidth > 100) {
      return spans[i];
    }
  }
}));
expect(wideElement.getText()).toEqual('Three');
```

11.4.7　链式调用定位操作

对于复杂的页面，元素往往不能一次性定位成功，需要逐层从上而下进行查找。基于element和element.all，可以轻松地在Protractor里实现对元素的链式定位。这个功能非常实用，可避免书写大量冗余的代码对元素对象进行迭代比较。常用的链式定位如表11-3所示。

表11-3　常用的链式定位

定 位 方 式	定位函数	说　　明
element(locator)	element	在父元素中查找一个子元素
element(locator)	all	在父元素中查找一组元素
element.all(locator)	all	在一组元素中查找一组符合条件的元素
element.all(locator)	filter	接受回调函数作为参数进行筛选，返回一组符合条件的元素
element.all(locator)	get	基于索引值返回某个元素
element.all(locator)	first	返回一组元素中的第一个元素
element.all(locator)	last	返回一组元素中的最后一个元素

以下示例中，首先找到所有class为parent的div元素，然后筛选class为foo的子元素，之后应该返回1a和2a。

被测HTML代码如下：

```
<div id='id1' class="parent">
  <ul>
    <li class="foo">1a</li>
    <li class="baz">1b</li>
  </ul>
</div>
<div id='id2' class="parent">
  <ul>
    <li class="foo">2a</li>
    <li class="bar">2b</li>
  </ul>
</div>
```

定位代码如下：

```
element.all(by.css('.parent')).all(by.css('.foo'));
```

以下示例通过filter在所有的列表元素里匹配文字。

被测HTML代码如下：

```
<ul class="items">
  <li class="one">First</li>
  <li class="two">Second</li>
  <li class="three">Third</li>
</ul>
```

定位代码如下：

```
element.all(by.css('.items li')).filter(function(elem, index) {
  return elem.getText().then(function(text) {
    return text === 'Third';
  });
}).first();
```

11.4.8 使用$和$$

在前端应用中，$是jQuery的一个别称，可以通过它构造一个jQuery对象，是广大前端工程师经常使用到的技巧。Protractor贴心地把$和$$引进到了全局变量，让测试人员可以使用与jQuery一致的语法结构，基于CssSelector进行一个或一组对象的定位。其中，$用于定位一个元素，$$则用于定位一组元素。

被测HTML代码如下：

```
<div class="parent1">
  <div class="child">
    <div>{{person.phone}}</div>
  </div>
```

```
</div>
<div class="parent2">
  <div class="child">
    <div>{{person.phone}}</div>
  </div>
</div>
```

定位代码如下：

```
$('.parent1').$('.child').element(by.binding('person.phone'));
```

以上定位代码的效果与以下代码相同，但更简洁明了。一般而言，链式定位表达式通常比较长，使用$和$$可以有效提高代码的可读性。

```
element(by.css('.parent1')).element(by.css('.child')).element(by.binding('person.phone'))
```

以下为$$的示例代码，用于定位一组li元素：

```
<div class="parent">
  <ul>
    <li class="one">First</li>
    <li class="two">Second</li>
    <li class="three">Third</li>
  </ul>
</div>
```

定位代码如下：

```
$('.parent').$$('li');
```

11.4.9　自定义定位策略

Protractor的强大之处在于它不仅提供了丰富的定位策略，还允许开发人员创建自己的定位策略。大型应用往往基于范式规划而成，遵循一定的设计思路与模式，充分利用这些

特殊性，创建自定义的定位策略往往能获得事半功倍的效果。

被测HTML代码如下：

```
<button ng-click="doAddition()">Go!</button>
```

添加自定义定位策略如下：

```
by.addLocator('buttonTextSimple',
    function(buttonText, opt_parentElement, opt_rootSelector) {
  // This function will be serialized as a string and will execute in the
  // browser. The first argument is the text for the button. The second
  // argument is the parent element, if any.
  var using = opt_parentElement || document,
      buttons = using.querySelectorAll('button');
  // Return an array of buttons with the text.
  return Array.prototype.filter.call(buttons, function(button) {
    return button.textContent === buttonText;
  });
});
```

定位代码如下：

```
element(by.buttonTextSimple('Go!'));
```

11.5 异步流程控制

请读者回忆一下第10章C#测试脚本的主要代码逻辑：

（1）打开必应网页。

（2）首先找到搜索框。

（3）设置搜索文字。

（4）找到搜索按钮。

（5）提交搜索内容。

这种调用方式是同步的，也就是说在执行下一步操作之前，上一步操作必须完成。实际上不仅C#，包括Java和Python都是用的以下这种同步方式进行浏览器操作的。

```
driver.Navigate().GoToUrl("http://www.bing.com");
// locate the search box by id, set "selenium" as search text
driver.FindElement(By.Id("sb_form_q")).SendKeys("selenium");
// locate the search button and click
driver.FindElement(By.Id("sb_form_go")).Click();
```

与同步方式不同，JavaScript使用了异步编程模型，这种编程模型基于回调函数实现事件驱动，避免了多线程模型同步的诸多复杂性，而且可以充分利用单核CPU，达到不错的性能表现。但是如果具体的业务逻辑比较复杂，就会出现大量的嵌套回调函数，大大降低代码的可读性和可维护性，也就是编程中的金字塔厄运（Pyramid Of Doom[①]）。尽管对这些回调函数采取分别命名和分离存放的措施可以在形式上减少嵌套代码的规模，但仍然不能有效降低嵌套的层数，无法从根本上解决问题[②]。

为了从根本上解决异步回调的金字塔问题，开源社区最早提出了Promise异步编程模式，它是对回调函数的抽象，力图在保留原有异步模型优势的情况下解决回调函数的嵌套问题。Promise本质是一个对象，用来传递异步操作的消息，它代表一个异步操作的执行结果，这个任务既可能已经完成，也可能仍在进行中。

Promise对象有3种状态：Pending（进行中）、Fulfilled（已完成）和 Rejected（已失败），只有异步操作的结果可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。

Promise模式的核心是调用then方法，它可以用来注册当promise完成或者失败时调用的回调函数。通过在then的函数体内返回一个新的promise对象，可以支持异步的链式调用，从而将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。

11.5.1　使用Promise

在Protractor中定位某个元素后，即可以对该元素进行多种操作，例如sendKeys（键入

---

① Devin Weaver. The Pyramid of Doom: A javaScript Style Trap[OL]. 2012. http://web.archive.org/web/20151209151711/http://tritarget.org/blog/2012/11/28/the-pyramid-of-doom-a-javascript-style-trap.

② Max Ogden. Callback Hell[OL]. [2016]. http://callbackhell.com/.

字符）、click（单击）等。所有这些操作都基于异步执行，因为底层的WebDriverJS使用了Promise[1]。注意，虽然WebDriverJs的Promise实现了基于CommonJS的Promise/A[2]提议，但并不完全遵守所有的规则。

接下来，仍然以本章第1节中用的ToDoList为例介绍Protractor处理底层的异步处理机制。以下测试代码通过对then的调用把若干个执行体串联起来，完成对DOM元素的操作以及断言，包括：

（1）打开网页。

（2）通过by.model('todoList.todoText')找到输入框。

（3）设置输入框文本为“first script”。

（4）通过by.css('[value="add"]')找到按钮。

（5）单击按钮。

（6）通过by.repeater('todo in todoList.todos')找到列表。

（7）使用expect(items.length).toEqual(3)断言列表子元素个数为3。

（8）获取列表第2个子元素文本。

（9）使用expect(itemtext).toEqual('first script')断言元素文本为“first script”。

```
describe('todo list', function() {
  it('should add a todo', function() {
   browser.get('/')
   .then(function () {
      return element(by.model('todoList.todoText'));
   })
   .then(function (val) {
      return val.sendKeys('first script');
   })
   .then(function () {
      return element(by.css('[value="add"]'));
   })
```

① SeleniumHQ. Understanding the Promise Manager[OL]. [2016]. https://github.com/SeleniumHQ/selenium/wiki/WebDriverJs#promises.

② CommonJS. Promises/A[OL]. [2016]. http://wiki.commonjs.org/wiki/Promises/A.

```
    .then(function (val) {
      return val.click();
    })
    .then(function () {
        element.all(by.repeater('todo in todoList.todos'))
        .then(function(items){
          expect(items.length).toEqual(3);
          items[2].getText()
          .then(function (itemtext) {
            expect(itemtext).toEqual('first script');
          });
        })
    })
  });
});
```

以上代码编写时一切都很顺利，代码逻辑也很清晰，但遗憾的是运行时却产生了如下错误：

```
Error: 'expect' was used when there was no current spec, this could be because an asynchronous
test timed out
```

该错误的原因是虽然WebDriverJs操作虽然已经时序化，但该测试用例用的是Jasmine框架。在实际执行中，Jasmine已经结束运行但WebDriverJs的异步操作还没有返回，所以没有任何断言发生。为了解决这个问题，可以用第4章介绍的Jasmine异步技巧，确保在断言执行之前不要退出测试用例，示例代码如下：

```
describe('todo list', function() {
  it('should add a todo', function(done) {
    browser.get('/')
    .then(function () {
      return element(by.model('todoList.todoText'));
```

```
    })
    .then(function (val) {
      return val.sendKeys('first script');
    })
    .then(function () {
      return element(by.css('[value="add"]'));
    })
    .then(function (val) {
      return val.click();
    })
    .then(function () {
        element.all(by.repeater('todo in todoList.todos'))
        .then(function(items){
          expect(items.length).toEqual(3);
          items[2].getText()
          .then(function (itemtext) {
            expect(itemtext).toEqual('first script');
            done();
          });
        })
     })
  });
});
```

11.5.2 定制的ControlFlow

上一节中介绍的Jasmine异步技巧结合使用WebDriverJs的Promise虽然解决了回调函数的嵌套问题，但代码量仍然不小。仅仅是查找对象、单击按钮、进行断言就用了三十多行代码，如果是更复杂的业务逻辑岂不是刚摆脱回调函数又要被then语句所包围？

为了解决该问题，WebDriverJs使用了一个定制化的Promise，它内部以ControlFlow为

核心对执行的异步操作进行时序控制，无需在测试代码中显式地添加then语句，也无需具体考虑把一系列异步操作进行串联代码实现，即可实现异步操作的顺序执行。从而，开发人员能够把主要精力放在具体业务逻辑上。那么ControlFlow是如何做到顺序控制的呢？

ControlFlow的完整模型由任务和任务队列组成。任务是ControlFlow中执行的最小单元，每个任务都会通过ControlFlow的execute函数进行时序安排，execute返回的对象正是一个Promise用于承载执行结果，如以下源代码所示。Protractor脚本中的元素操作正是以任务的形式被执行。若干个任务从属于某个任务队列，在JavaScript事件循环的轮转中得到执行。

```
execute(fn, opt_description) {
  if (isGenerator(fn)) {
    let original = fn;
    fn = () => consume(original);
  }
  if (!this.hold_) {
    var holdIntervalMs = 2147483647;  // 2^31-1; max timer length for Node.js
    this.hold_ = setInterval(function() {}, holdIntervalMs);
  }
  var task = new Task(
      this, fn, opt_description || '<anonymous>',
      {name: 'Task', top: ControlFlow.prototype.execute});
  var q = this.getActiveQueue_();
  q.enqueue(task);
  this.emit(ControlFlow.EventType.SCHEDULE_TASK, task.description);
  return task.promise;
}
```

在execute函数创建完成任务后，该任务会被压入当前的任务列表内。如果任务列表不存在，则进行创建，如以下源代码所示：

```
getActiveQueue_() {
    if (this.activeQueue_) {
```

```
      return this.activeQueue_;
    }
    this.activeQueue_ = new TaskQueue(this);
    if (!this.taskQueues_) {
      this.taskQueues_ = new Set();
    }
    this.taskQueues_.add(this.activeQueue_);
    this.activeQueue_
        .once('end', this.onQueueEnd_, this)
        .once('error', this.onQueueError_, this);
    asyncRun(() => this.activeQueue_ = null);
    this.activeQueue_.start();
    return this.activeQueue_;
}
```

既然元素操作在ControlFlow中以任务的形式被执行，那随后的回调函数如何保证能够以期望的时序执行呢？WebDriverJS会把脚本中的回调部分转换为一个then任务也放到任务队列中，从而满足按序执行的要求。如果读者对ControlFlow的完整实现感兴趣，可以参考网址https://github.com/SeleniumHQ/selenium/blob/master/javascript/node/selenium-webdriver/lib/promise.js中的源码。

注意，Protractor的定位操作返回的是ElementFinder或ElementArrayFinder对象。虽然它们可以理解为对WebElement的包装，但不同之处在于ElementFinder对象不会立即根据指定的Locator来查找到页面上的元素，ElementFinder只在调用了元素对象的操作方法时，它才会真正实施查找执行操作。这种定位延迟绑定的优点是声明某个元素的时候并不需要该元素必须存在。复杂的页面一般元素很多，而且这些元素不一定在页面刚加载的时候就全部存在，延迟绑定可以实现在统一的地方对这些元素进行声明，这对设计良好的测试代码帮助很大。本书将在第13章介绍利用这一优点实现页面对象模型的方法。

现在，把上一节的测试代码基于ControlFlow重新改写如下。可以发现代码清晰简单了很多。

```
describe('todo list', function() {
```

```
  it('should add a todo', function() {
    browser.get('/');
    element(by.model('todoList.todoText')).sendKeys('first script');
    element(by.css('[value="add"]')).click();
    var todoList = element.all(by.repeater('todo in todoList.todos'));
    expect(todoList.count()).toEqual(3);
    expect(todoList.get(2).getText()).toEqual('first script');
  });
});
```

11.5.3　JavaScript测试框架的异步适配器

ControlFlow实现了通过同步代码执行异步的DOM操作，但Jasmine本身并不理解WebDriverJS的ControlFlow，那么如何保证Jasmine的断言也能够被按序执行呢？为解决该问题，Protractor提供了Jasmine和WebDriverJs之间的适配器jasminewd2[①]（随同Protractor被默认安装），它包含以下功能：

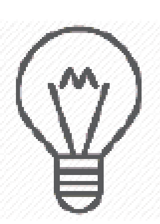
（1）把Jasmine原生的函数基于ControlFlow进行了包装，然后替换了原生函数。当ControlFlow的所有任务完成前，任何一个测试用例都会保持等待状态而不会退出。以下代码是部分被重新包装的Jasmine函数。

```
global.it = wrapInControlFlow(global.it, 'it');
global.fit = wrapInControlFlow(global.fit, 'fit');
global.beforeEach = wrapInControlFlow(global.beforeEach, 'beforeEach');
global.afterEach = wrapInControlFlow(global.afterEach, 'afterEach');
global.beforeAll = wrapInControlFlow(global.beforeAll, 'beforeAll');
global.afterAll = wrapInControlFlow(global.afterAll, 'afterAll');
```

（2）Jasmine的expect操作会把断言作为一个任务压入ControlFlow中，从而保证断言按序执行。

（3）断言执行之前首先获得Promise对象返回的结果。

① Protractor Adaptions. Protractor[OL]. [2016]. http://www.protractortest.org/#/control-flow.



如果读者使用其他的测试框架进行Protractor测试，请注意其是否有对应的异步适配器。目前Jasmine、Mocha和Cucumber都有各自的适配器实现，如果读者选择的测试框架还没有对应的适配器，则需要自行编写，或者基于各个测试框架自身对异步操作的支持编写测试代码。

11.6 页面交互

Protractor提供了丰富的页面交互API，绝大多数是对WebDriverJs既有方法的包装。本节将列出经常使用的部分。

11.6.1 操作浏览器

在Protractor中可以通过全局变量browser访问浏览器对象，类型为ProtractorBrowser。因为browser是一个对WebDriver对象的包装，也可以先通过browser.driver获得WebDriver对象，然后直接调用WebDriverJs提供的方法。

如以下源代码所示，ProtractorBrowser在构造函数里通过以下代码实现了WebDriverJs方法的重定向，所以也可以通过browser调用WebDriverJs的方法，以达到精简代码的目的。

```
Object.getOwnPropertyNames(webdriver.WebDriver.prototype)
    .forEach(function (method) {
    if (!_this[method] &&
        typeof webdriverInstance[method] == 'function') {
        if (methodsToSync.indexOf(method) !== -1) {
            ptorMixin(_this, webdriverInstance, method, _this.waitForAngular.bind(_this));
        }
        else {
            ptorMixin(_this, webdriverInstance, method);
        }
    }
```

```
        });

function ptorMixin(to, from, fnName, setupFn) {

    to[fnName] = function () {

        for (var i = 0; i < arguments.length; i++) {

            if (arguments[i] instanceof element_1.ElementFinder) {

                arguments[i] = arguments[i].getWebElement();

          }

        }

        if (setupFn) {

            setupFn();

        }

        return from[fnName].apply(from, arguments);

    };

}
```

以下为常用的浏览器操作函数：

- browser.get

该函数的作用是令浏览器打开目标页面，功能与browser.driver.get一致。

- browser.refresh

该函数的作用是重新加载当前网页。

- browser.actions

该函数的作用是建立一个操作序列，所有操作只有在调用了perform后才被执行。示例代码如下：

```
browser.actions().

    mouseDown(element1).

    mouseMove(element2).

    mouseUp().

    perform();
```

- browser.touchActions

该函数的作用是建立一个触摸操作序列，只有在调用了perform后才被执行。示例代

码如下：

```
browser.touchActions().
    tap(element1).
    doubleTap(element2).
    perform();
```

- browser.executeScript

该函数的作用是在当前窗口内执行一段JavaScript代码，可以引用当前窗口内的有效变量。如果需要传递参数给该段JavaScript代码，可使用arguments对象。示例代码如下：

```
var el = element(by.id('header'));
var tag = browser.executeScript('return arguments[0].tagName', el);
expect(tag).toEqual('h1');
```

- browser.getPageSource

该函数的作用是获得当前页面的源码。

- browser.getCurrentUrl

该函数的作用是获得当前页面的URL。

- browser.getTitle

该函数的作用是获得当前页面的标题。

- browser.takeScreenshot

该函数的作用是对整个页面或当前窗口进行截屏，保存为base64编码的PNG格式。

- browser.switchTo

该函数的作用是在多个frame或窗口之间切换焦点。示例代码如下：

```
browser.switchTo().frame(element(by.tagName('iframe')).getWebElement());
```

- browser.getCapabilities

该函数的作用是获得当前的浏览器配置信息。

- browser.getAllWindowHandles

该函数的作用是获得所有窗口的窗口句柄，返回值可供browser.switchTo使用。

11.6.2　操作元素

以下为常用的元素操作函数：

- element(locator).isPresent

该函数的作用是检查元素是否在页面上存在。示例代码如下：

```
expect(element(by.binding('person.name')).isPresent()).toBe(true);
```

- element(locator).click

该函数的作用是单击当前元素。示例代码如下：

```
element(by.partialLinkText('Doge')).click();
```

- element(locator).sendKeys

该函数的作用是在当前元素上键入文字或命令。

- element(locator).getTagName

该函数的作用是获得当前元素的tag名称。示例代码如下：

```
expect(element(by.binding('person.name')).getTagName()).toBe('span');
```

- element(locator).getAttribute

该函数的作用是获得当前元素的属性。示例代码如下：

```
<div id="foo" class="bar"></div>

var foo = element(by.id('foo'));

expect(foo.getAttribute(class)).toEqual('bar');
```

- element(locator).getText

该函数的作用是获得当前元素的可见文字，包括子元素部分。

- element(locator).getSize

该函数的作用是获得当前元素的大小，以像素为单位。

- element(locator).getLocation

该函数的作用是获得当前元素的位置。

- element(locator).isEnabled

该函数的作用是判断当前元素是否是启用状态。

- element(locator).submit

该函数的作用是提交表单，可以是表单内的元素或者是表单元素本身。示例代码如下：

```
<form id="login">
  <input name="user">
</form>
var login_form = element(by.id('login'));
login_form.submit();
```

- element(locator).clear

该函数的作用是清除当前元素的值，只对input和textarea元素有效。示例代码如下：

```
<input id="foo" value="Default Text">
var foo = element(by.id('foo'));
expect(foo.getAttribute('value')).toEqual('Default Text');
foo.clear();
expect(foo.getAttribute('value')).toEqual('');
```

- element(locator).isDisplayed

该函数的作用是判断当前元素是否为可视状态。示例代码如下：

```
<div id="foo" style="visibility:hidden">
var foo = element(by.id('foo'));
expect(foo.isDisplayed()).toBe(false);
```

- element(locator).takeScreenshot

该函数的作用是为当前元素截屏。示例代码如下：

```
function writeScreenShot(data, filename) {
  var stream = fs.createWriteStream(filename);
  stream.write(new Buffer(data, 'base64'));
  stream.end();
}
```

```
var foo = element(by.id('foo'));

foo.takeScreenshot().then((png) => {

  writeScreenShot(png, 'foo.png');

});
```

11.7　Protractor的等待机制

一个网站应用不仅有静态的HTML元素，也有根据用户的操作动态修改的页面显示。有些操作会有比较明显的延时，例如有时前端Web应用需要先从远端的服务器获取数据，然后再把数据渲染到页面上。这种情况下，Protractor在查找元素之前需要等待足够的时间来确保JavaScript的动作已经完成，否则查找元素会失败。根据应用场景的不同，Protractor提供了不同的等待机制。

11.7.1　waitForAngular

AngularJS有自己独特的，区别于其他前端框架的$digest循环机制，该循环结束后完成页面渲染。为了充分利用这个机制，Protractor提供了waitForAngular，这个函数的作用是等待当前正在执行的$digest、$http或者$timeout完成。在Protractor执行元素查找之前会先把waitForAngular作为任务压入到ControlFlow中，从而确保所有的网络、渲染以及异步操作完成后再查找对应的元素。基于Protractor的AngularJS测试使开发人员可以将主要精力集中在业务逻辑上，而不用编写大量的显式等待代码去确保元素已经在页面上存在。

为了充分发挥waitForAngular的作用，建议开发AngularJS应用的时候，尽量使用AngularJS原生的$http、$timeout等服务，而不要用JavaScript的相应函数。



Protractor框架已经在各元素操作内部调用了waitForAngular，测试脚本里不应该再显式调用该函数。

某些AngularJS应用可能出于业务要求会不间断地调用$http访问远程服务，对于这种应用的测试，需要禁止waitForAngular，否则测试代码会一直处于等待状态无法继续执行。waitForAngular可以通过以下代码禁用，这也是测试非AngularJS应用的必要步骤，但

开发人员需要自己添加额外的等待机制保证测试代码的正确执行。

```
browser.ignoreSynchronization = true;
```

11.7.2 使用sleep

对于非AngularJS的应用，当需要等待一段时间（例如等某个元素出现在页面上）再执行下一句脚本命令时，使用sleep是最简便的方式。这是一个WebDriverJs提供的函数，参数为等待时间，单位是毫秒。以下示例代码让测试脚本等待5秒。

```
browser.sleep(5000);
```

虽然browser.sleep简单易用，但缺点也很明显。测试脚本之所以要进行等待，往往是需要等某些异步调用完成或元素状态的更新，在不同的网络环境和系统配置下，这个时间是不一定的，甚至无法预料的。为了保证测试在绝大多数环境下可以顺利进行，测试人员往往会根据经验赋予其一个较长的等待时间。在这样的设计下，就算页面已经提前完成状态更新，sleep仍然会按照指定参数继续等待，这样的测试效率很低，不推荐大量使用。

11.7.3 隐式等待

隐式等待是一个全局配置，设置了隐式等待后，接下来的每次元素定位都有一个超时限制。在达到这个超时限制之前，如果WebDriver没有在页面中找到期望的元素，则会基于一个时间周期不断循环检查元素状态。如果在超时前找到了期望的元素，WebDriver会提前退出等待继续执行脚本。如果超时限制到了却仍然找不到期望的元素，则抛出找不到元素的异常。以下示例代码设置了隐式等待时间为5秒。

```
browser.driver.manage().timeouts().implicitlyWait(5000);
```

隐式等待默认为关闭状态，在实际脚本开发中，可能需要根据脚本执行的阶段不同，在同一个测试用例中多次调用implicitlyWait调整等待时间。如果设置时间为0，则关闭隐式等待。由于隐式等待会基于一个时间周期循环执行元素查找代码，较长的超时限制或较慢的定位方式会影响测试效率，因此不建议设置太久的隐式等待时间。

11.7.4　显式等待

显式等待是一种使用很广泛的等待机制，它能够指定具体需要等待的元素状态以及所超时时间。如果某个页面操作比较耗时，则使用太长的隐式等待对测试效率影响较大。针对这种情况，使用显式等待能够对不同的元素有针对性地进行设置，灵活性更高。

显式等待通过向browser.wait传入等待条件得以实现。例如以下示例代码通过自定义函数urlChanged来判断当前的Url是否符合期望的http://www.bing.com，如果5秒后仍然不符合等待条件则抛出异常。

```
var urlChanged = function() {
  return browser.getCurrentUrl().then(function(url) {
    return url === 'http://www.bing.com';
  });
};
browser.wait(urlChanged, 5000);
button.click();
```

除了使用自定义的函数进行条件匹配，也可以通过全局变量ExpectedConditions获得Protractor内置的匹配条件。以下示例代码在单击按钮之前执行了显式等待，保证在单击按钮之前，该按钮已经进入了可单击状态。

```
var button = $('#mybutton');
var isClickable = ExpectedConditions.elementToBeClickable(button);
browser.wait(isClickable, 5000); //wait for the button to be clickable
button.click();
```

其他常用的内置等待条件包括：

- alertIsPresent

该条件用于判断是否出现期望的警告框。示例代码如下：

```
browser.wait(ExpectedConditions.alertIsPresent(), 5000);
```

- textToBePresentInElement

该条件用于判断元素的文本内是否包含所指定的字串。示例代码如下：

```
browser.wait(ExpectedConditions.textToBePresentInElement($('#abc'), 'foo'), 5000);
```

- titleContains

该条件用于判断页面标题是否包括所指定的字串，匹配模式大小写敏感。示例代码如下：

```
browser.wait(ExpectedConditions.titleContains('foo'), 5000);
```

- urlContains

该条件用于判断页面URL内是否包含所指定的字串，匹配模式大小写敏感。示例代码如下：

```
browser.wait(ExpectedConditions.urlContains('foo'), 5000);
```

- presenceOf

该条件用于判断期望所指定的元素在页面中存在，但并不需要为可见状态。示例代码如下：

```
browser.wait(ExpectedConditions.presenceOf($('#abc')), 5000);
```

- stalenessOf

该条件用于判断期望所指定的元素在页面中不存在，与presenceOf的作用相反。示例代码如下：

```
browser.wait(ExpectedConditions.stalenessOf($('#abc')), 5000);
```

- visibilityOf

该条件用于判断期望所指定的元素属于可见状态，意味着该元素有非0的高宽。示例代码如下：

```
browser.wait(ExpectedConditions.visibilityOf($('#abc')), 5000);
```

- invisibilityOf

该条件用于判断期望所指定的元素属于不可见状态，与visibilityOf作用相反。示例代码如下：

```
browser.wait(ExpectedConditions.invisibilityOf($('#abc')), 5000);
```

- elementToBeSelected

该条件用于判断所指定的选择框元素为选中状态。示例代码如下：

```
browser.wait(ExpectedConditions.elementToBeSelected($('#myCheckbox')), 5000);
```

ExpectedConditions支持匹配条件的布尔运算包括not、and和or。例如以下代码表示页面标题包含Foo，同时不能为FooBar。

```
var titleContainsFoo = ExpectedConditions.titleContains('Foo');

var titleIsNotFooBar = ExpectedConditions.not(EC.titleIs('FooBar'));

browser.wait(ExpectedConditions.and(titleContainsFoo, titleIsNotFooBar), 5000);
```

11.8　测试非AngularJS程序

Protractor同时支持测试AngularJS和非AngularJS应用，但编写非AngularJS测试脚本时，测试人员需要使用到上一节提到的隐式等待或显式等待，从而保证脚本的执行时序。本节以一个实例演示测试非AngularJS程序需要注意的地方。

（1）创建本地文件夹NonAngular，并添加以下被测index.html文件。

```
<!DOCTYPE html>

<html lang="en">

<head>

   <meta charset="UTF-8">

   <title>NonAngularJS</title>

</head>

<body>

   <input type="button" id="mybutton" onclick="onClick()" value="Click Me"></input>

   <script>

         function onClick(){

            setTimeout(onTimeoutShort, 500);
```

```
            setTimeout(onTimeoutLong, 3000);
        };
        function onTimeoutShort(){
            var shortdiv = document.createElement('div');
            shortdiv.id = 'shortdiv';
            shortdiv.innerHTML = 'Implicit Wait';
            document.body.appendChild(shortdiv);
        };
        function onTimeoutLong(){
            var longdiv = document.createElement('div');
            longdiv.id = 'longdiv';
            longdiv.innerHTML = 'Explicit Wait';
            document.body.appendChild(longdiv);
        }
    </script>
</body>
</html>
```

（2）启动命令控制台，执行命令http-server，通过浏览器访问http://localhost:8080，确保可以成功访问。

（3）启动另一个命令控制台，执行以下命令：

```
npm install -g protractor
npm init
npm install protractor --save-dev
node .\node_modules\protractor\node_modules\webdriver-manager update
```

（4）在页面上单击Click Me按钮会触发两个延时操作，分别是1秒和3秒。第1个延时操作会在按钮单击1秒后为页面添加一个id为shortdiv的对象，文本为Implicit Wait，第2个延时操作会在按钮单击3秒后为页面添加一个id为longdiv的对象，文本为Explicit Wait。

（5）编写以下测试代码模拟单击按钮行为，并验证这两个延时操作是否工作正常。

local.conf.js：

```
exports.config = {
  directConnect:true,
  specs: ['index.spec.js'],
  baseUrl: 'http://localhost:8080',
  framework: 'jasmine2'
};
```

index.sepc.js：

```
describe('NonAngular Test', function() {
  it('should wait for timer', function() {
    browser.get('/');
    $('#mybutton').click();
    expect($('#shortdiv').getText()).toBe('Implicit Wait');
    expect($('#longdiv').getText()).toBe('Explicit Wait');
  });
});
```

（6）启动另一个命令控制台，执行命令protractor local.conf.js，返回错误信息“Angular could not be found on the page http://localhost:8080/ : retries looking for angular exceeded”。其原因是Protractor默认设置为测试AngularJS应用，否则会超时退出。为了让Protractor测试非AngularJS应用，需在加载页面前进行如下设置：

```
browser.ignoreSynchronization = true
```

（7）再次运行测试代码，以上错误消失，但测试用例会返回新的错误信息“No element found using locator: By(css selector, #shortdiv)”，原因是shortdiv会在1秒延时后才被添加到页面中，但测试代码立即进行了断言而并没有等足1秒，这时候shortdiv还不存在。

（8）为测试代码添加隐式等待如下，在定位shortdiv前隐式等待1秒，从而确保shortdiv可以被成功定位。

```
describe('NonAngular Test', function() {
```

```
  it('should wait for timer', function() {
    browser.ignoreSynchronization = true;
    browser.driver.manage().timeouts().implicitlyWait(1000);
    browser.get('/');
    $('#mybutton').click();
    expect($('#shortdiv').getText()).toBe('Implicit Wait');
    expect($('#longdiv').getText()).toBe('Explicit Wait');
  });
});
```

（9）除了shortdiv，longdiv会在3秒后才会被添加到页面中，但3秒时间较长不适合使用隐式等待。这种情况下，可以使用显式等待实时检查longdiv的状态，修改后的最终测试代码如下：

```
describe('NonAngular Test', function() {
  it('should wait for timer', function() {
    browser.ignoreSynchronization = true;
    browser.driver.manage().timeouts().implicitlyWait(1000);
    browser.get('/');
    $('#mybutton').click();
    expect($('#shortdiv').getText()).toBe('Implicit Wait');
    browser.wait(ExpectedConditions.presenceOf($('#longdiv')), 5000);
    expect($('#longdiv').getText()).toBe('Explicit Wait');
  });
})
```

第12章 使用Selenium Server

如第11章示例代码所示，Protractor对Chrome提供了本地直连的功能，使开发人员在脚本编写和本地调试阶段有非常便捷的开发体验。但另一方面，一个功能完善的自动化测试框架还需要能够覆盖包括Firefox、IE和Edge等在内的其他多种浏览器。这些浏览器版本众多，运行在不同的操作系统上也可能产生不同的测试结果。本章将重点介绍用Protractor实现多浏览器测试的技术细节，这也是下一章分布式测试的基础。

本章将介绍：

- Selenium Server环境配置
- JSON Wire Protocol与W3C WebDriver标准
- Selenium 3.0
- 配置浏览器

12.1 Selenium Server环境配置

第11章的配置示例展示了Protractor通过指定directConnect来实现对Chrome浏览器的直连操作。示例代码如下：

```
exports.config = {
  directConnect:true,
  specs: ['todo-spec.js'],
  baseUrl: 'http://localhost:8080',
  framework: 'jasmine2'
};
```

除了Chrome，Protractor也支持对Firefox的直连操作，如图12-1所示，但仅限于直连47版本及其之前的Firefox。本书将在12.4.2节详细解释其原因和解决方案。

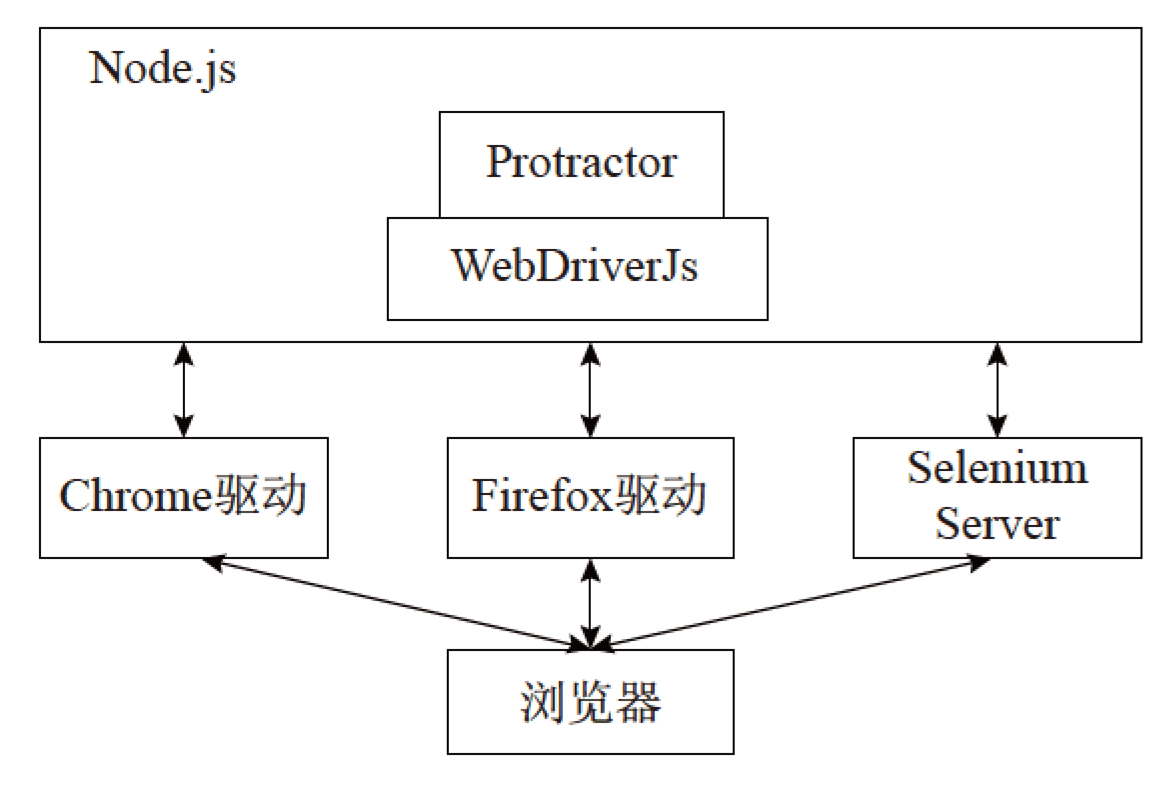
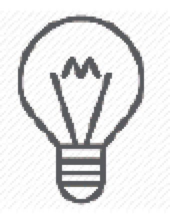


图12-1　驱动浏览器

而其他类型的浏览器则需要Selenium Server间接完成操作，因为并不是每种浏览器都在技术上支持跨机器的远程调用。在这种情况下，为了满足以下需求，需要Selenium Server充当远程调用中转的角色与被测浏览器运行在相同的机器上。

- 统一的接口兼顾多种浏览器和版本
- 统一的接口兼顾多种操作系统和版本
- 统一的接口实现分布式测试

测试代码只需要将命令发送到远程Selenium Server即可，对于浏览器而言仍然是本地调用。



Selenium Server并不等同于Selenium RC的HTTP代理，Selenium Server的主要作用是以统一的接口支持远程调用。

12.1.1　安装Java JDK

Selenium Server依赖于Java运行环境，在运行Selenium Server前需要先安装Java JDK（Java SE Development Kit），可访问http://www.oracle.com/technetwork/java/javase/downloads/index.html了解更多信息。作者撰写本书时JDK的最新版本为8u112，如图12-2所示。

根据操作系统选择对应的JDK安装包，保持默认选项进行安装。安装完成后，在命令控制台中输入java并按回车键，如果显示Java的用法介绍，则说明JDK安装成功。

Java SE Development Kit 8u112

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux x86	162.42 MB	jdk-8u112-linux-i586.rpm
Linux x86	177.12 MB	jdk-8u112-linux-i586.tar.gz
Linux x64	159.97 MB	jdk-8u112-linux-x64.rpm
Linux x64	174.73 MB	jdk-8u112-linux-x64.tar.gz
Mac OS X	223.15 MB	jdk-8u112-macosx-x64.dmg
Solaris SPARC 64-bit	139.78 MB	jdk-8u112-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.06 MB	jdk-8u112-solaris-sparcv9.tar.gz
Solaris x64	140.46 MB	jdk-8u112-solaris-x64.tar.Z
Solaris x64	96.86 MB	jdk-8u112-solaris-x64.tar.gz
Windows x86	188.99 MB	jdk-8u112-windows-i586.exe
Windows x64	195.13 MB	jdk-8u112-windows-x64.exe

图12-2　Java SE安装包

12.1.2　下载Selenium Server Standalone

Selenium Server Standalone 是运行Selenium Server的载体文件，启动Selenium Server前需要先下载Selenium Server Standalone。

Protractor依赖于WebDriver管理工具webdriver-manager，下载Protractor后可以直接在命令控制台执行以下命令下载Selenium Server Standalone：

```
node .\node_modules\protractor\node_modules\webdriver-manager update
```

下载到的Selenium Server Standalone二进制文件位于node_modules/protractor/node_modules/webdriver-manager/selenium文件夹内。

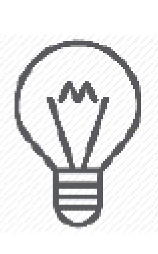
如果使用全局Protractor运行测试用例，则可以执行以下命令将其下载到全局目录：

```
webdriver-manager update
```

在作者撰写本书时，webdriver-manager默认下载的是Selenium 2里最新的selenium-server-standalone-2.53.1.jar，这由webdriver-manager的配置文件node_modules/protractor/node_modules/webdriver-manager/built/config.json所决定的。

```
  "webdriverVersions": {
    "selenium": "2.53.1",
```

```
  "chromedriver": "2.25",
  "geckodriver": "v0.11.1",
  "iedriver": "2.53.1",
  "androidsdk": "24.4.1",
  "appium": "1.6.0"
}
```



2016年10月，Selenium社区发布了Selenium 3.0，当前最新的Selenium Server Standalone二进制文件为selenium-server-standalone-3.0.1.jar。如果读者希望调整webdriver-manager的默认下载版本，可以修改config.json中的selenium字段。

除了webdriver-manager管理工具，也可以直接访问https://selenium-release.storage.googleapis.com/查询所有有效的Selenium Server版本及其下载地址，例如访问https://selenium-release.storage.googleapis.com/2.53/selenium-server-standalone-2.53.1.jar下载2.53.1版本的二进制文件。

12.1.3 下载浏览器驱动

除了Selenium二进制文件，webdriver-manager update命令也可以用于下载浏览器驱动，它默认会下载Chrome对应的chromedriver和Firefox对应的geckodriver，同样保存于本地文件夹node_modules/protractor/node_modules/webdriver-manager/selenium中。在命令控制台下执行以下命令会同时下载IE的32位和64位驱动：

```
node .\node_modules\protractor\node_modules\webdriver-manager update --ie --ie32
```

除了通过webdriver-manager下载浏览器驱动，也可以直接访问对应的网站分别进行下载，将在12.4节详细介绍。

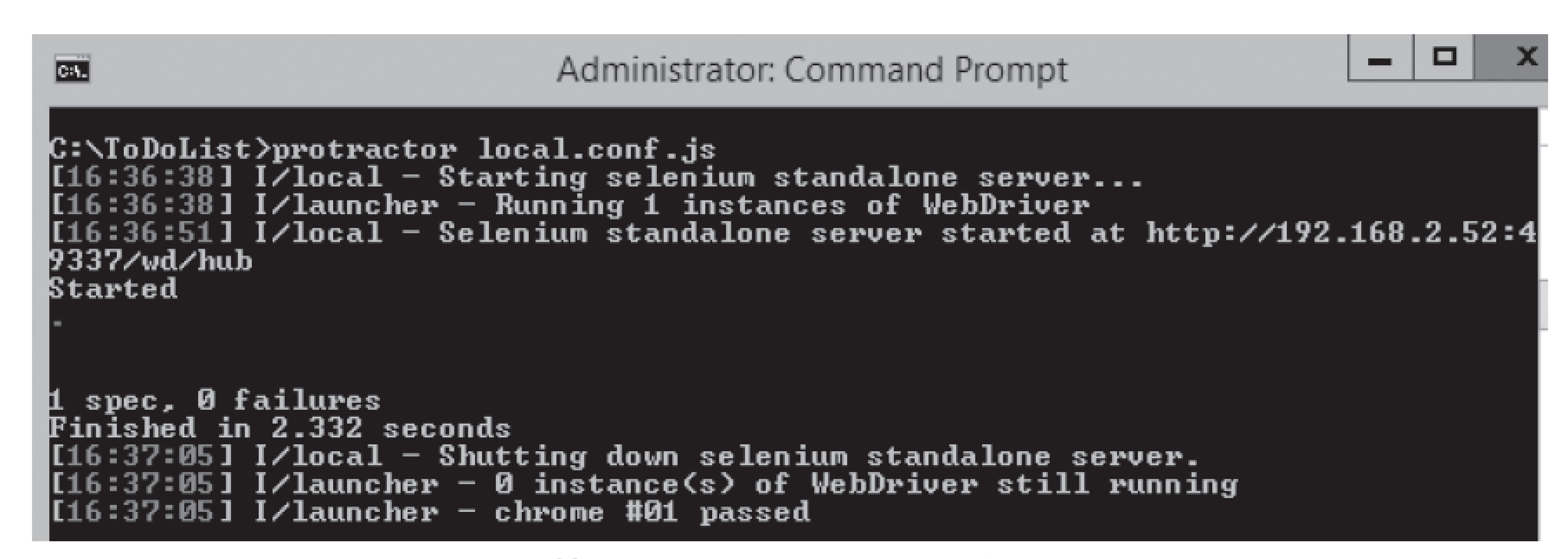
12.1.4 配置Protractor

在Protractor配置文件内设置directConnect为false（默认值）。运行测试用例时，

Protractor会自动在当前机器启动Selenium Server。在此之前，应先确保已经通过webdriver-manager update下载了Selenium Server Standalone二进制文件和Chrome驱动。设置代码如下：

```
exports.config = {
  directConnect:false,
  specs: ['todo-spec.js'],
  baseUrl: 'http://localhost:8080',
  capabilities:{
    browserName:'chrome'
  },
  framework: 'jasmine2'
};
```

当测试结束后，Selenium Server将自动停止，如图12-3所示。


```
C:\ToDoList>protractor local.conf.js
[16:36:38] I/local - Starting selenium standalone server...
[16:36:38] I/launcher - Running 1 instances of WebDriver
[16:36:51] I/local - Selenium standalone server started at http://192.168.2.52:4
9337/wd/hub
Started
.

1 spec, 0 failures
Finished in 2.332 seconds
[16:37:05] I/local - Shutting down selenium standalone server.
[16:37:05] I/launcher - 0 instance(s) of WebDriver still running
[16:37:05] I/launcher - chrome #01 passed
```

图12-3　基于Selenium Server运行Protractor

以上这种配置非常适合测试人员在脚本开发或修改过程中，迅速地进行本地验证而无需依赖于专门的Selenium Server服务器。

12.1.5　启动Selenium Server

除了让Protractor代为启动Selenium Server，开发人员也可以手工启动一个Selenium Server并让Protractor测试用例运行其上。该Selenium Server既可以运行在本机，也可以运行在远程机器中。

如图12-4所示，在命令控制台下执行webdriver-manager start命令，即可启动Selenium Server，它默认运行在4444端口。

```
Administrator: Command Prompt - webdriver-manager  start

C:\>webdriver-manager start
[16:54:15] I/start - java -Dwebdriver.chrome.driver=C:\Users\Administrator\AppData\Roaming\npm\node_
modules\protractor\node_modules\webdriver-manager\selenium\chromedriver_2.25.exe -Dwebdriver.gecko.d
river=C:\Users\Administrator\AppData\Roaming\npm\node_modules\protractor\node_modules\webdriver-mana
ger\selenium\geckodriver-v0.11.1.exe -jar C:\Users\Administrator\AppData\Roaming\npm\node_modules\pr
otractor\node_modules\webdriver-manager\selenium\selenium-server-standalone-2.53.1.jar -port 4444
[16:54:15] I/start - seleniumProcess.pid: 5016
16:54:16.094 INFO - Launching a standalone Selenium Server
16:54:16.153 INFO - Java: Oracle Corporation 25.112-b15
16:54:16.153 INFO - OS: Windows Server 2012 R2 6.3 amd64
16:54:16.165 INFO - v2.53.1, with Core v2.53.1. Built from revision a36b8b1
16:54:16.230 INFO - Driver class not found: com.opera.core.systems.OperaDriver
16:54:16.230 INFO - Driver provider com.opera.core.systems.OperaDriver is not registered
16:54:16.242 INFO - Driver provider org.openqa.selenium.safari.SafariDriver registration is skipped:

registration capabilities Capabilities [{browserName=safari, version=, platform=MAC}] does not match
 the current platform WIN8
16:54:16.243 INFO - Driver class not found: org.openqa.selenium.htmlunit.HtmlUnitDriver
16:54:16.244 INFO - Driver provider org.openqa.selenium.htmlunit.HtmlUnitDriver is not registered
16:54:16.347 INFO - RemoteWebDriver instances should connect to: http://127.0.0.1:4444/wd/hub
16:54:16.348 INFO - Selenium Server is up and running
```

图12-4　用webdriver-manager start命令启动Selenium Server

修改Protractor配置脚本如下，seleniumAddress字段指向已经运行的Selenium Server地址。

```
exports.config = {
  directConnect:false,
  specs: ['todo-spec.js'],
  baseUrl: 'http://localhost:8080',
  seleniumAddress: 'http://192.168.2.52:4444/wd/hub',
  capabilities:{
    browserName:'chrome'
  },
  framework: 'jasmine2'
};
```

如图12-5所示，测试用例运行在指定的Selenium Server上。

```
Administrator: Command Prompt

C:\ToDoList>protractor local.conf.js
[17:01:27] I/hosted - Using the selenium server at http://192.168.2.52:4444/wd/h
ub
[17:01:27] I/launcher - Running 1 instances of WebDriver
Started
.


1 spec, 0 failures
Finished in 2.038 seconds
[17:01:33] I/launcher - 0 instance(s) of WebDriver still running
[17:01:33] I/launcher - chrome #01 passed
```

图12-5　运行测试用例

除了webdriver-manager，开发人员也可以直接在命令控制台调用java命令启动

Selenium Server，这比较适合手工下载Selenium Server Standalone二进制文件和各浏览器驱动的情况。如图12-6所示，分别通过Dwebdriver.chrome.driver和jar参数指定Chrome驱动和Selenium Server Standalone二进制文件的路径，并将Selenium Server运行在5000端口上。

```
Administrator: Command Prompt - java  -Dwebdriver.chrome.driver=c:\selenium\chromedriver_2.25.exe...

C:\selenium>java -Dwebdriver.chrome.driver=c:\selenium\chromedriver_2.25.exe -jar selenium-server-st
andalone-2.53.1.jar -port 5000
17:14:15.557 INFO - Launching a standalone Selenium Server
17:14:15.620 INFO - Java: Oracle Corporation 25.112-b15
17:14:15.620 INFO - OS: Windows Server 2012 R2 6.3 amd64
17:14:15.632 INFO - v2.53.1, with Core v2.53.1. Built from revision a36b8b1
17:14:15.695 INFO - Driver class not found: com.opera.core.systems.OperaDriver
17:14:15.696 INFO - Driver provider com.opera.core.systems.OperaDriver is not registered
17:14:15.704 INFO - Driver provider org.openqa.selenium.safari.SafariDriver registration is skipped:

registration capabilities Capabilities [{browserName=safari, version=, platform=MAC}] does not match
 the current platform WIN8
17:14:15.705 INFO - Driver class not found: org.openqa.selenium.htmlunit.HtmlUnitDriver
17:14:15.706 INFO - Driver provider org.openqa.selenium.htmlunit.HtmlUnitDriver is not registered
17:14:15.799 INFO - RemoteWebDriver instances should connect to: http://127.0.0.1:5000/wd/hub
17:14:15.799 INFO - Selenium Server is up and running
```

图12-6　用java命令启动Selenium Server

12.2　JSON Wire Protocol与W3C WebDriver标准

无论是使用Protractor通过Selenium Server远程操作浏览器，还是如第10章所述通过C#进行本地操作，技术基础都是Selenium社区主持的JSON Wire Protocol[①]协议。它统一了WebDriver的浏览器接口，定义了基于REST概念的服务端客户端通信协议。REST是一种分布式系统的实现风格，由Roy Fielding于2000年在他的博士论文中提出[②]，目前广泛的在各网络应用中得到了实现。

基于JSON Wire Protocol协议，Selenium社区对各浏览器驱动进行了服务端实现，而各种WebDriver的编程语言绑定则是客户端的实现。客户端与服务端基于JSON格式，通过HTTP以请求和响应的方式进行通信，把浏览器的操作命令传给服务端。



使用C#进行远程操作时，需要使用RemoteWebDriver，并将Selenium Server地址作为参数传入。

① SeleniumHQ. JsonWireProtocol[OL]. [2016]. https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol.

② Roy Thomas Fielding. Representational State Transfer[OL]. 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

随着WebDriver的迅速发展，Browser Testing and Tools工作组成立，成员来自于Microsoft、Google和Mozilla等公司，包括大名鼎鼎的WebDriver创建者Simon Stewart[1]。该工作组致力于将WebDriver API这套无关平台、无关语言的浏览器接口标准化，即现在的W3C WebDriver[2]，它是对JSON Wire Protocol的进一步完善与扩展。当前，W3C WebDriver标准仍然处于修订状态，但各厂商已经开始基于W3C标准对各自的浏览器驱动进行开发，例如Mozilla的geckodriver等。

W3C WebDriver和JSON Wire Protocol都是基于HTTP协议实现的，这意味着任何支持HTTP调用的编程语言都可以实现WebDriver的客户端绑定。对于开发人员而言，并不需要深入了解W3C WebDriver标准的每一个细节，但理解它的实现原理对以后项目中修改框架以及二次开发有很大帮助。

以下步骤通过Chrome的插件Postman，演示如何通过手工发送HTTP请求，模拟Protractor操作浏览器的过程。

（1）在http://192.168.2.51:4444上启动Selenium Server。

（2）如图12-7所示，在Postman中发送POST请求http://192.168.2.51:4444/wd/hub/session创建一个新的会话Session。每一个会话与某个浏览器对象对应，本示例通过在Body内指定browserName创建了一个Chrome对象的会话。Selenium Server接受到该请求后，返回JSON对象，并包含SessionId作为该会话的唯一标识。

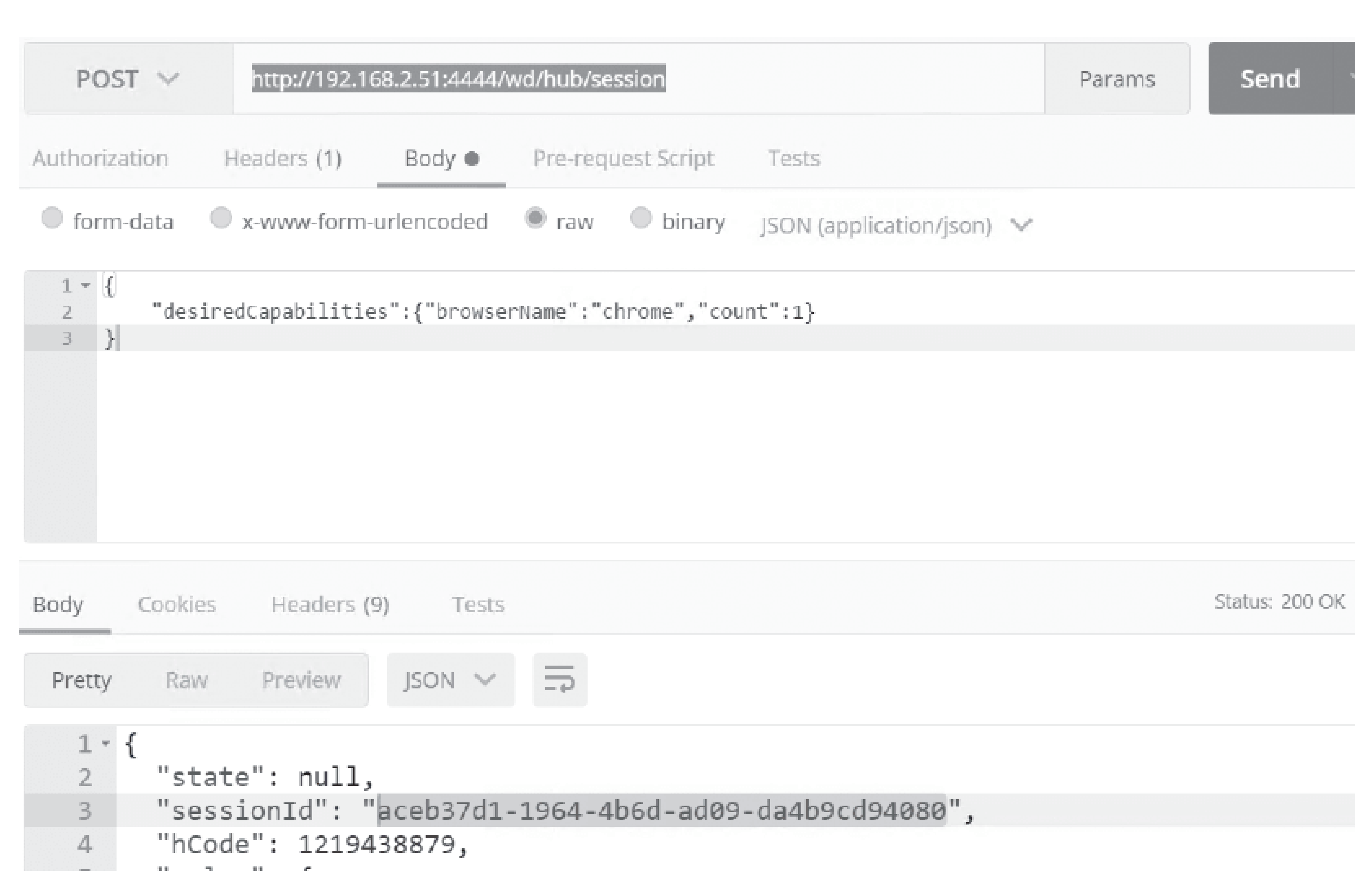


图12-7　创建会话

① W3C. Participants in the Browser Testing and Tools Working Group[OL]. [2016]. https://www.w3.org/2000/09/dbwg/details?group=49799&public=1.

② W3C. WebDriver[OL]. [2016]. https://www.w3.org/TR/webdriver/.

（3）发送POST请求http://192.168.2.51:4444/wd/hub/session/<SessionId>/url，其中，SessionId为上一步返回得到的会话标识id。如图12-8所示，在请求的Body内指定目标地址为http://www.bing.com，Selenium Server在浏览器内打开对应网站并返回成功的状态。

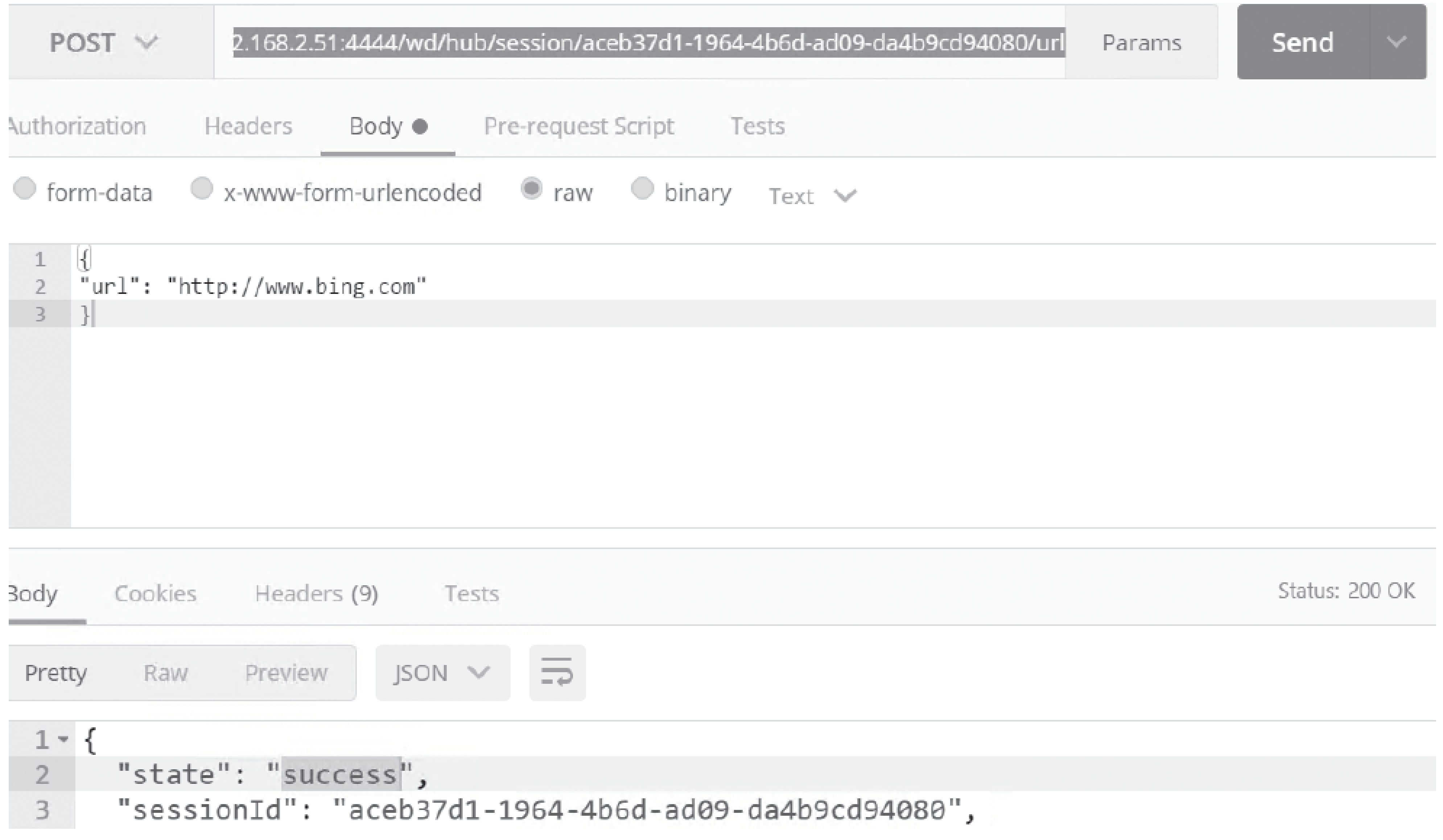


图12-8　打开网站

（4）发送GET请求http://192.168.2.51:4444/wd/hub/session/<SessionId>/title，获取已打开的网站的标题。如图12-9所示，Bing被成功返回。

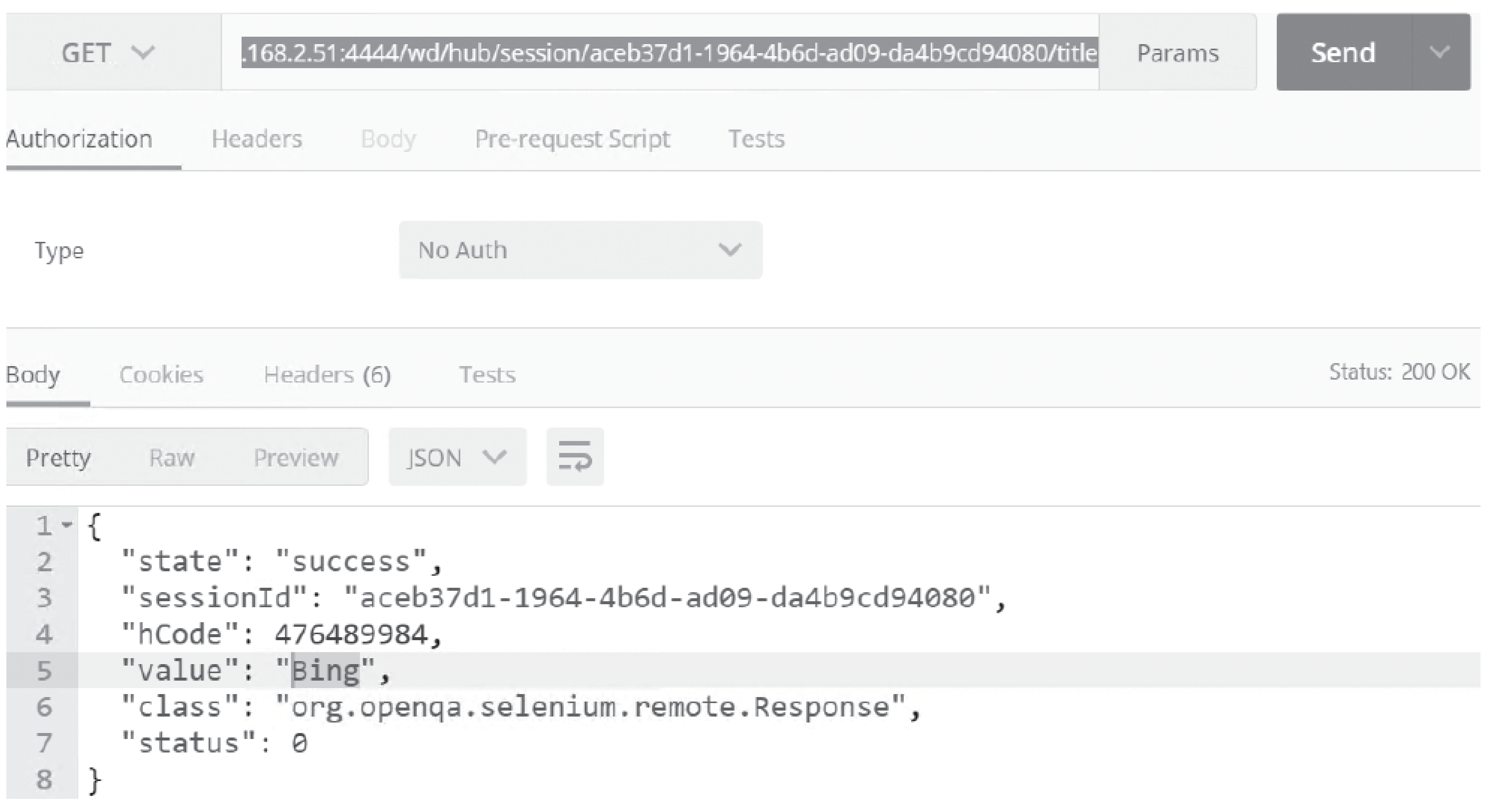


图12-9　返回标题

12.3 Selenium 3.0

2016年10月，Selenium社区发布了Selenium 3.0[①]。与Selenium 2.0相比，3.0版本最显著的变化是移除了基于JavaScript的Selenium Core和Selenium RC API，这意味着所有基于Selenium RC的测试将遇到兼容性问题，具体解决办法请参考https://seleniumhq.wordpress.com/2016/10/04/selenium-3-is-coming/中的论述。考虑到Selenium RC已经退出历史舞台，对于所有新自动化测试项目，强烈建议基于WebDrive开展测试工作。

对于已经在Selenium 2.0上进行WebDriver自动化测试的情况，由于Selenium 3.0并没有移除已有的WebDriver API，这些测试用例应该可以直接运行于Selenium 3.0上。不过考虑到3.0版刚刚发布，仍然存在缺陷修复的情况，如果读者遇到任何兼容性问题，建议关注Selenium 3.0对应的发布公告。在作者编写本书的时候，对应于Selenium 3.0的最新Selenium Server Standalone二进制文件是3.0.1版本，可以在网址https://selenium-release.storage.googleapis.com/2.53/selenium/selenium-server-standalone-3.0.1.jar处下载。在命令控制台执行以下命令即可启动Selenium Server 3，其他参数与2.0版一致。

```
java -jar selenium-server-standalone-3.0.1.jar
```

考虑到Selenium 3.0刚正式发布，而Protractor的默认下载版本仍然是Selenium 2.0，本书后续将仍然基于Selenium 2.0进行Protractor演示。

12.4 配置浏览器

如表12-1所示，Protractor对Chrome、Firefox、Safari、IE和Edge都有良好的支持。尽管PhantomJS也能够被Protractor所驱动，但因为PhantomJS的某些行为与真实的浏览器不一致，Protractor开发组不建议使用PhantomJS进行Protractor自动化测试。另外，目前Protractor还不支持Opera，如果项目中需要使用到Opera，建议使用其他测试框架

---

① SeleniumHQ. Selenium 3.0: Out Now![OL]. 2016. https://seleniumhq.wordpress.com/2016/10/13/selenium-3-0-out-now/.

和编程语言。

表12-1 Protractor对浏览器的支持

浏 览 器	是 否 支 持
Chrome	是
Firefox	是
Safari	是
Edge	是
IE	是
Opera	否
PhantomJS	不推荐

Protractor不仅可以在capabilities字段中通过browserName设置目标浏览器的名字，也可以对测试行为进行配制。例如以下代码将代理设置为http://localhost.com:8445/。

```
exports.config = {
  directConnect:true,
  capabilities: {browserName: chrome,
  proxy: {
       proxyType: 'manual',
       httpProxy: 'http://localhost.com:8445/'
     },
},
  specs: ['todo-spec.js'],
  baseUrl: 'http://localhost:8080'
};
```

表12-2为自动化测试中常用的浏览器capability的选项。

表12-2 常用浏览器capability的选项

名 称	类 型	描 述
browserName	string	浏览器的名字
version	string	浏览器的版本
platform	string	操作系统的名字

（续表）

名　称	类　　型	描　　述
logName	string	记录文件中显示的当前功能设置的名字，默认为浏览器的名字
count	number	属于当前配置选项的测试用例的运行次数，默认为1
shardTestFiles	string	相同配置选项下的测试文件是否可以并行执行，默认为否
maxInstances	number	当前配置选项下，能同时运行的浏览器个数
specs	string[]	只在当前配置选项下运行的测试文件
exclude	string[]	不在当前配置选项下运行的测试文件
databaseEnabled	boolean	Session是否可以与数据库存储交互
applicationCacheEnabled	boolean	Session是否可以与应用的缓存交互
webStorageEnabled	boolean	Session是否可以与Web缓存交互
acceptSslCerts	boolean	Session是否默认接受所有的SSL证书
proxy	proxy JSON object	设置代理
unexpectedAlertBehaviour	string	浏览器应该如何处理unhandled exception，可以设置为accept、dismiss或者ignore

大型应用的测试用例往往很多，结合使用shardTestFiles和maxInstances可以大幅提高测试效率。例如以下配置文件中包含两个测试文件，设置shardTestFiles为true后允许这两个测试文件中的用例以并行的方式运行在不同的Chrome浏览器内，如图12-10所示，有两个Chrome浏览器对象同时被启动。

```
exports.config = {
  directConnect:false,
  specs: ['todo-spec.js','todo-spec2.js'],
  baseUrl: 'http://localhost:8080',
  capabilities:{
    browserName:'chrome',
    shardTestFiles:true,
    maxInstances:2
  }
};
```

除了以上共同的属性，不同的浏览器还支持各自独有的配置选项，读者可以参考网址https://github.com/SeleniumHQ/selenium/wiki/DesiredCapabilities中的内容。

```
Administrator: Command Prompt
C:\ToDoList>protractor local.conf.js
[23:31:36] I/launcher - Running 2 instances of WebDriver
..[23:32:10] I/testLogger -
------------------------------------

[23:32:10] I/testLogger - [chrome #01-1] PID: 672
[chrome #01-1] Specs: C:\ToDoList\todo-spec2.js
[chrome #01-1]
[chrome #01-1] [23:31:37] I/local - Starting selenium standalone server...
[chrome #01-1] [23:31:51] I/local - Selenium standalone server started at http://192.168.2.52:65488/
wd/hub
[chrome #01-1] Started
[chrome #01-1] .
[chrome #01-1]
[chrome #01-1]
[chrome #01-1] 1 spec, 0 failures
[chrome #01-1] Finished in 2.001 seconds
[chrome #01-1] [23:32:10] I/local - Shutting down selenium standalone server.

[23:32:10] I/testLogger -

[23:32:10] I/launcher - 1 instance(s) of WebDriver still running
[23:32:10] I/testLogger -
------------------------------------

[23:32:10] I/testLogger - [chrome #01-0] PID: 4608
[chrome #01-0] Specs: C:\ToDoList\todo-spec.js
[chrome #01-0]
[chrome #01-0] [23:31:37] I/local - Starting selenium standalone server...
[chrome #01-0] [23:31:51] I/local - Selenium standalone server started at http://192.168.2.52:54930/
wd/hub
[chrome #01-0] Started
[chrome #01-0] .
[chrome #01-0]
[chrome #01-0]
[chrome #01-0] 1 spec, 0 failures
[chrome #01-0] Finished in 1.982 seconds
[chrome #01-0] [23:32:10] I/local - Shutting down selenium standalone server.

[23:32:10] I/testLogger -

[23:32:10] I/launcher - 0 instance(s) of WebDriver still running
[23:32:10] I/launcher - chrome #01-1 passed
[23:32:10] I/launcher - chrome #01-0 passed
```

图12-10　并行执行测试脚本

12.4.1　Chrome

Protractor默认使用Chrome驱动测试用例，在进行测试之前，需要确保Chrome驱动已经被下载到本地。执行以下命令即可启动支持Chrome的Selenium Server，当然，读者也可用webdriver-manage启动Selenium Server。

```
java -Dwebdriver.chrome.driver=c:\selenium\chromedriver_2.25.exe -jar selenium-server-standalone-2.53.1.jar -port 5000
```

在以上命令中，Chrome驱动被放置于c:\selenium下，并通过Dwebdriver.chrome.driver参数指定驱动所在路径。

12.4.2　Firefox

对于Firefox 47及其之前的版本，用于Selenium自动化测试的Firefox驱动由Selenium社区通过Firefox插件的形式提供，该插件已经被包含在了Selenium包内，测试开始后会被自

动加载。所以，对于Firefox 47及其之前版本的自动化测试，无论是C#、Java还是Protractor都无需专门下载浏览器驱动，这一点是区别于其他浏览器的地方。

与Chrome类似，在Protractor中通过设置directConnect即可直接连接到Firefox，示例配置如下：

```
exports.config = {
  directConnect:true,
  specs: ['todo-spec.js'],
  baseUrl: 'http://localhost:8080',
  capabilities:{
    browserName:'firefox'
  },
  framework: 'jasmine2'
};
```

同样，无需指定驱动路径，以下命令启动Selenium Server后即直接支持Firefox 47：

```
java -jar selenium-server-standalone-2.53.1.jar
```

Mozilla于2016年6月推出Firefox 48，该版本带来的重大变化是无论正式发布版还是测试版，任何未签名的Firefox插件都无法安装[①]。所以，从Firefox 48开始，Selenium社区的驱动插件不再适用，Protractor也无法直连版本为48之后的Firefox。

为了解决针对Firefox 48之后版本的自动化测试问题，Mozilla基于W3C WebDriver标准开发了geckodriver用于支持Gecko浏览器包括Firefox的自动化测试。与其他浏览器驱动类似，geckodriver需要单独下载，在作者编写本书时它的最新版本为0.11.1，可以从网址https://github.com/mozilla/geckodriver/releases处获得。



请注意，引入geckodriver的直接原因是Firefox的版本及其设计发生了变化，与Selenium 3无关。

以下命令使用Selenium 2启动Selenium Server并支持Firefox 48及以后的版本：

① Mozilla. Add-ons/Extension Signing[OL]. [2016]. https://wiki.mozilla.org/Addons/Extension_Signing.

```
java -jar selenium-server-standalone-2.53.1.jar -Dwebdriver.gecko.driver=c:\selenium\
geckodriver-v0.11.1.exe
```

geckodriver基于Mozilla的Marionette①自动化协议实现，为了让Protractor自动化脚本能够运行在Selenium 2之上的Selenium Server，需要在配置文件中启用Marionette，代码如下：

```
exports.config = {
  directConnect:false,
  seleniumAddress: 'http://192.168.2.52:4444/wd/hub',
  specs: ['todo-spec.js'],
  baseUrl: 'http://localhost:8080',
  capabilities:{
    browserName: 'firefox',
    version: '50.0.2',
    marionette: 'true'
  },
  framework: 'jasmine2'
};
```

由于Selenium 3默认启用了Marionette，因此无需在Protractor配置文件中进行声明。

```
exports.config = {
  directConnect:false,
  seleniumAddress: 'http://192.168.2.52:4444/wd/hub',
  specs: ['todo-spec.js'],
  baseUrl: 'http://localhost:8080',
  capabilities:{
    browserName: 'firefox',
    version: '50.0.2'
```

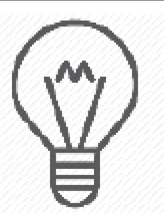
① Mozilla. Marionette[OL]. [2016]. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Marionette.

```
  },
  framework: 'jasmine2'
};
```

以下命令使用Selenium 3启动Selenium Server并支持Firefox 48及以后的版本：

```
java -jar selenium-server-standalone-3.0.1.jar -Dwebdriver.gecko.driver=c:\selenium\
geckodriver-v0.11.1.exe
```

当前geckodriver还没有正式发布，仍处于频繁的功能更新和缺陷修复阶段，Protractor官方建议仍然以Firefox 47为主要的测试对象[1]，读者后续可以在https://github.com/mozilla/geckodriver持续关注geckodriver的开发进展。



老版本的Firefox可以从网址https://ftp.mozilla.org/pub/firefox/releases/处下载，建议安装后取消自动升级，否则Firefox会自动升级到最高版本。

12.4.3 Edge

Edge浏览器的驱动由微软提供，可以从网址https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/处下载。它同时支持W3C WebDriver标准和JSON Wire Protocol，以实现向后兼容。下载前，请确保浏览器驱动与操作系统版本号保持一致。

以下命令用于启动Selenium Server并支持Edge：

```
java -jar selenium-server-standalone-2.53.1.jar -Dwebdriver.edge.driver=c:\selenium\
MicrosoftWebDriver.exe
```

Protractor配置文件声明测试对象为Edge浏览器，代码如下：

```
exports.config = {
  directConnect:false,
  seleniumAddress: 'http://192.168.2.52:4444/wd/hub',
  specs: ['todo-spec.js'],
```

① Protractor. Browser Support[OL]. [2016]. https://github.com/angular/protractor/blob/f9c8a37f7dbec1dccec2dde0bd6884ad7ae3f5c7/docs/browser-support.md.

```
  baseUrl: 'http://localhost:8080',
  capabilities:{
    browserName: 'MicrosoftEdge',
    elementScrollBehavior: 1,
    nativeEvents: false
  },
  framework: 'jasmine2'
};
```

12.4.4 IE

IE浏览器的驱动可以从网址http://docs.seleniumhq.org/download/处下载。由于实际环境下32位的IE使用居多，而在64位的浏览器驱动下输入框操作性能很慢，建议读者使用32位的IE浏览器驱动进行自动化测试。

以下命令用于启动Selenium Server并支持IE：

```
java -jar selenium-server-standalone-2.53.1.jar -Dwebdriver.ie.driver=c:\selenium\
IEDriverServer.exe
```

Protractor配置文件声明测试对象为IE浏览器，代码如下：

```
exports.config = {
  directConnect:false,
  seleniumAddress: 'http://192.168.2.52:4444/wd/hub',
  specs: ['todo-spec.js'],
  baseUrl: 'http://localhost:8080',
  capabilities:{
    browserName: 'internet explorer'
  },
  framework: 'jasmine2'
};
```

IE 11是目前微软唯一支持的IE版本，为保证在IE 11上正常运行测试用例，请确保在进行测试前完成以下设置：

（1）启动IE 11，执行菜单命令View→Zoom，设置缩放比例为100%，如图12-11所示。

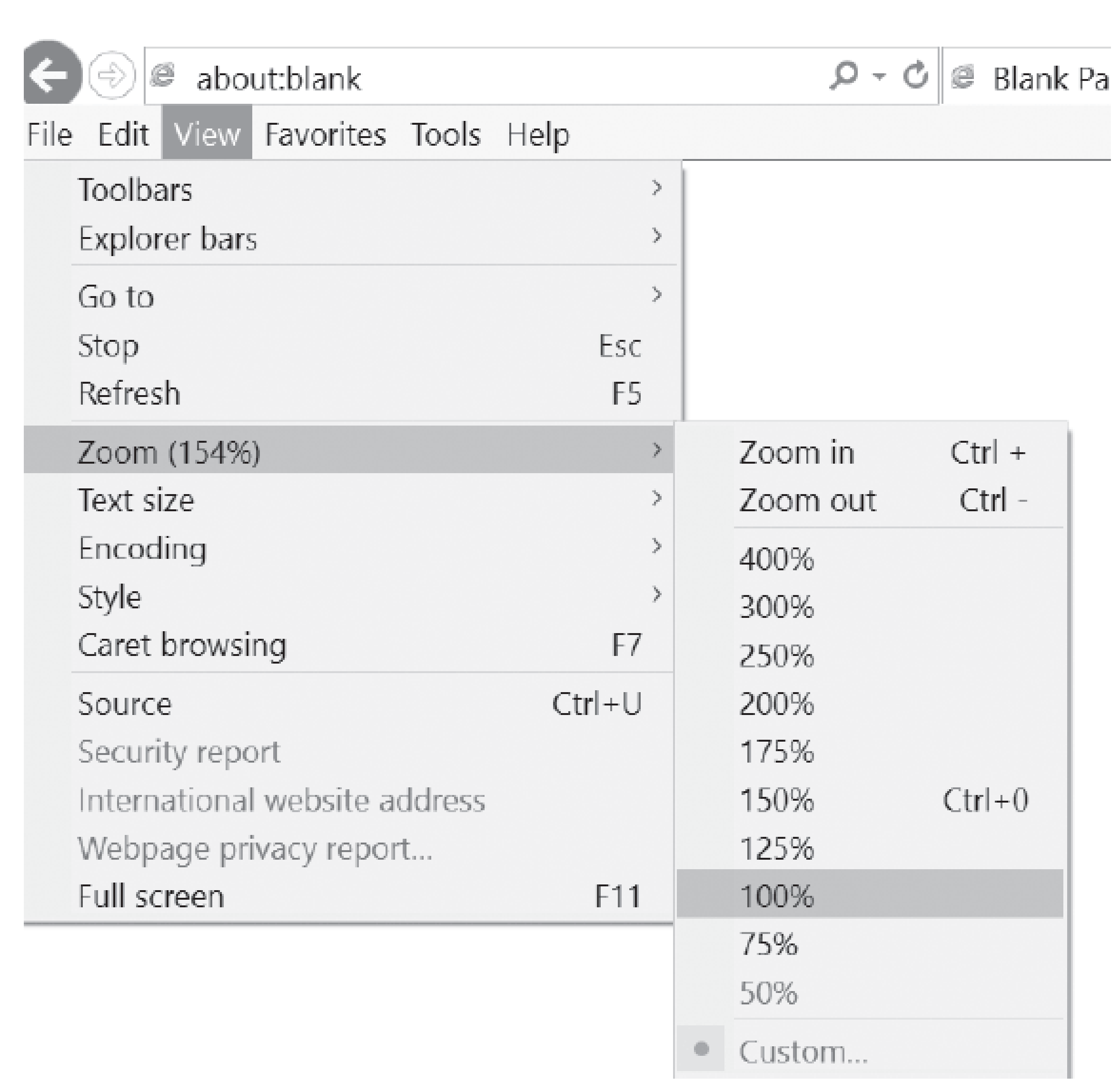


图12-11　设置IE的缩放比例

（2）执行菜单命令Tools→Internet Options，在弹出的对话框中选择Security选项卡，确保Internet、Local intranet、Trusted sites和Restricted sites都取消勾选Enable Protected Mode复选框，如图12-12所示。

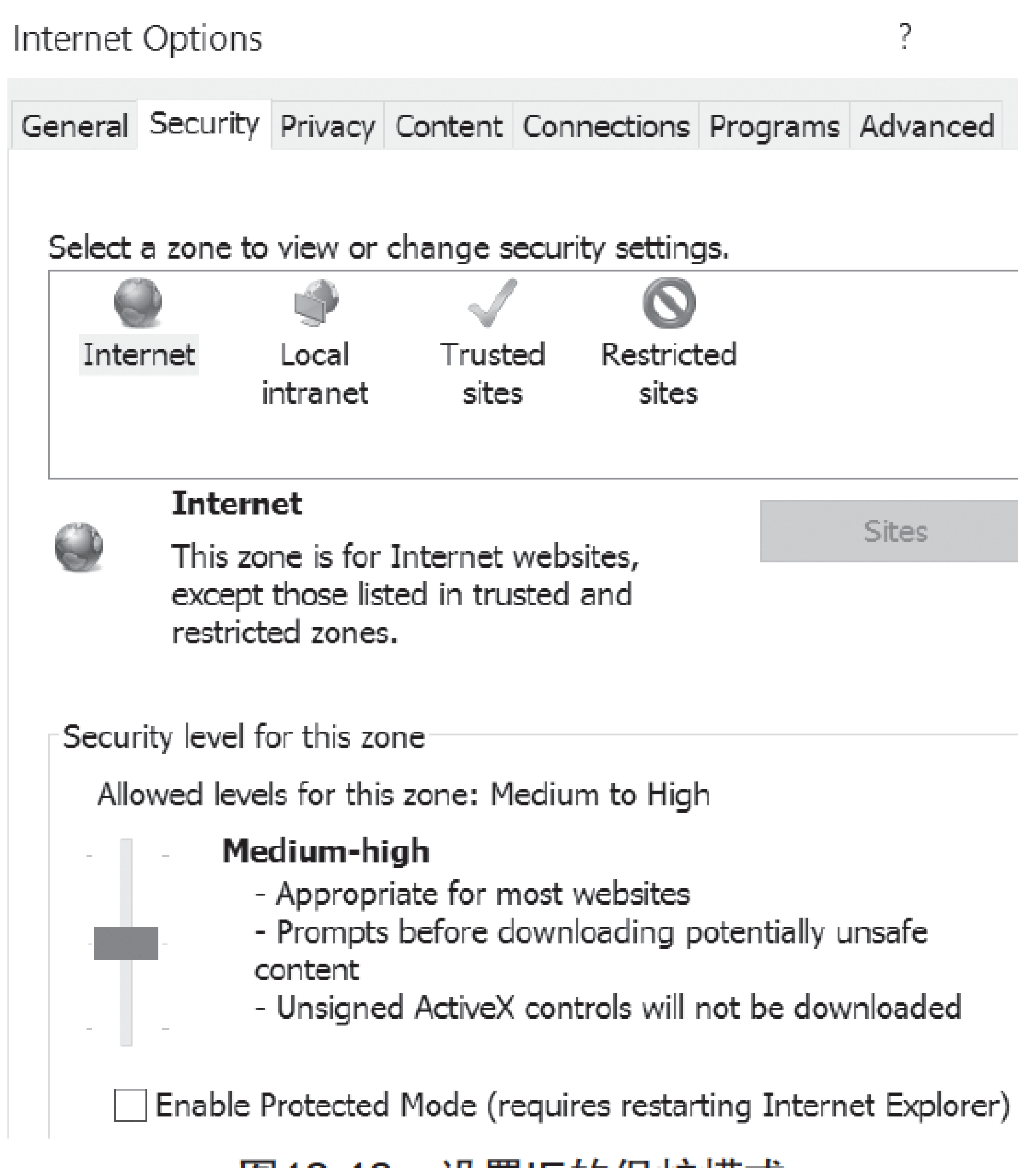


图12-12　设置IE的保护模式

（3）执行菜单命令Tools→Internet Options，在弹出的对话框中选择Advanced选项卡，取消勾选Enable Enhanced Protected Mode复选框，如图12-13所示。

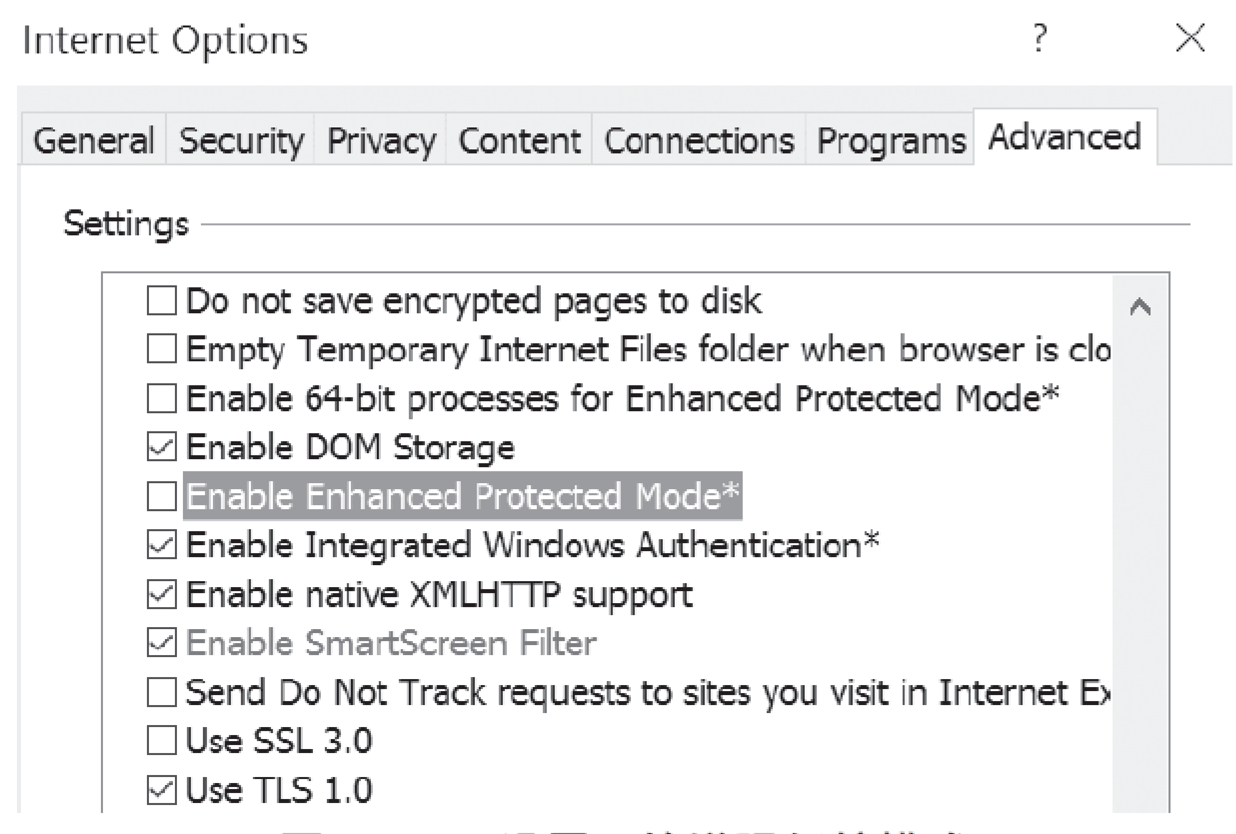


图12-13　设置IE的增强保护模式

（4）对于32位系统，确保注册表键HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\Main\FeatureControl\FEATURE_BFCACHE存在，创建一个DWORD类型的键，名字为iexplore.exe，值为0。对于64位系统，确保注册表键HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Internet Explorer\Main\FeatureControl\FEATURE_BFCACHE存在，创建一个DWORD类型的键，名字为iexplore.exe，值为0。

12.4.5　多浏览器测试

在启动Selenium Server的时候，通过参数指定不同浏览器的驱动，该Selenium Server可以同时支持多种浏览器的测试。例如以下Selenium Server同时支持IE、Edge和Chrome。

```
java -jar selenium-server-standalone-2.53.1.jar

 -Dwebdriver.ie.driver=c:\selenium\IEDriverServer.exe

 -Dwebdriver.edge.driver=c:\selenium\MicrosoftWebDriver.exe

 -Dwebdriver.chrome.driver=c:\selenium\chromedriver_2.25.exe
```

为了避免命令行参数过长，可以把驱动所在文件夹设置到Path环境变量内，如图12-4所示。

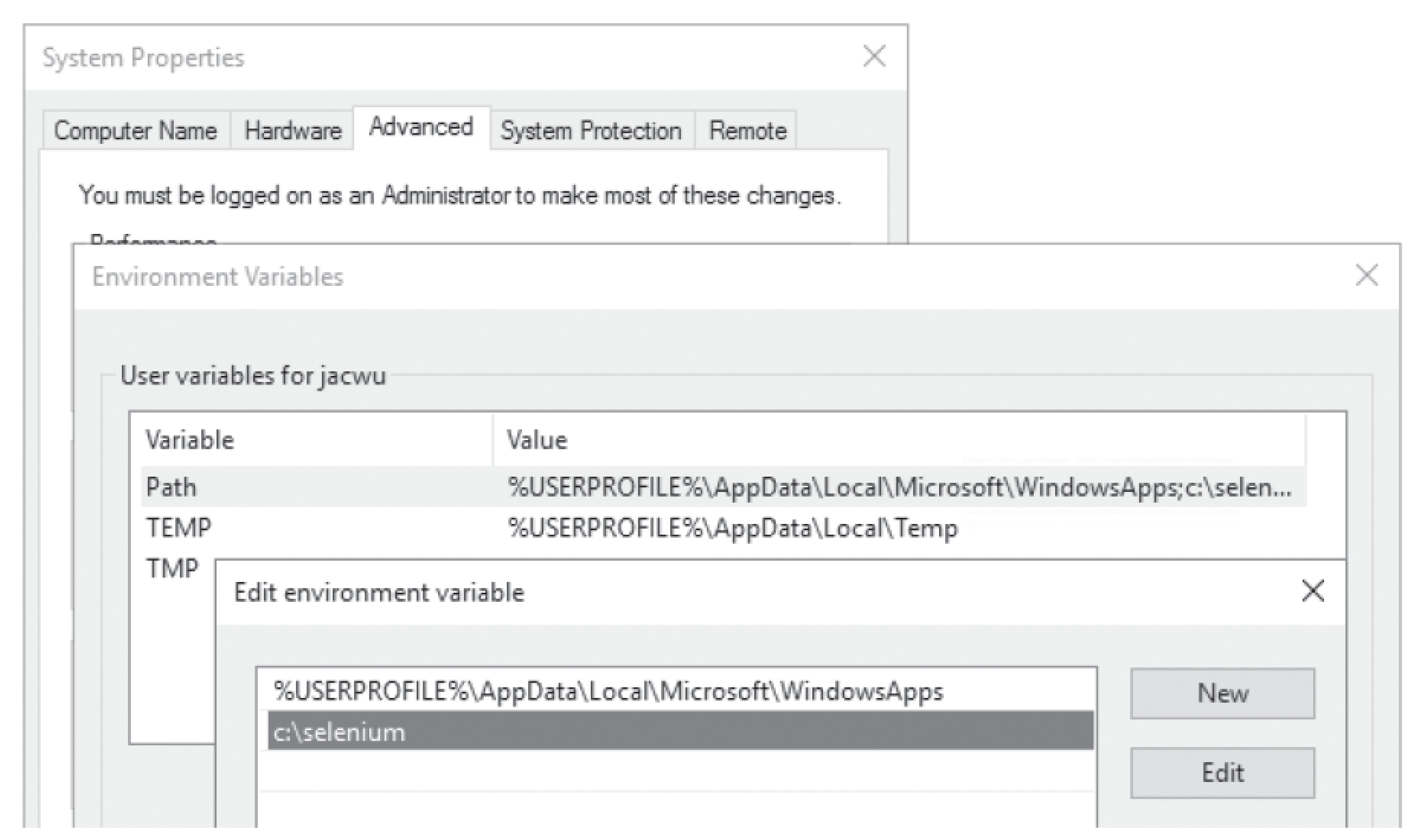


图12-14　添加环境变量

添加了环境变量后，命令无需再指定驱动全路径，Selenium Server会根据名称在Path环境变量内自动匹配对应的驱动。

```
java -jar selenium-server-standalone-2.53.1.jar
 -Dwebdriver.ie.driver=IEDriverServer.exe
 -Dwebdriver.edge.driver=MicrosoftWebDriver.exe
 -Dwebdriver.chrome.driver=chromedriver_2.25.exe
```

为了让Protractor测试用例能够同时在多种浏览器上运行，需要使用multiCapabilities指定各个目标浏览器的名字。以下示例代码同时支持Firefox、Chrome和IE。

```
exports.config = {
  directConnect:false,
  seleniumAddress: 'http://192.168.2.52:4444/wd/hub',
  specs: ['todo-spec.js'],
  baseUrl: 'http://localhost:8080',
  multiCapabilities: [
    {browserName: 'firefox'},
    {browserName: 'chrome'},
    {browserName: 'internet explorer'}],
```

```
  framework: 'jasmine2'
};
```

读者可能还记得第10章的C#示例中通过工厂模式创建的浏览器对象。比较而言，Protractor通过配置文件指定浏览器的方式更为方便、灵活，任何时候如果要增加或减少对某种浏览器的支持，只需要修改配置文件即可，无需任何额外的编译过程。

第13章 自动化测试最佳实践

Protractor为用户提供了自动化测试框架，那么如何编写复用率高、维护成本低的测试代码呢？本章将基于Protractor介绍自动化测试中的最佳实践，探讨如何开发一个高复用度的测试框架。

本章将介绍：

- 页面对象模型
- 数据驱动测试
- 测试报告
- 性能测试
- 图像匹配
- 任务自动化

13.1 页面对象模型

结合Jasmine对测试用例进行组织，Protractor提供了完整的自动化测试解决方案，这是不是就意味着开发人员就一定可以基于Protractor进行低成本、高效率的自动化测试呢？答案是否定的。

自动化与人工测试最大的区别就是通过代码驱动测试，这是自动化的优势所在，但也往往是很多自动化测试失败的根源。特别是大型项目，随着页面功能不断增多，页面交互变得复杂，开发人员不得不反复投入大量精力重新编写已经写好的测试代码，以满足新的需求调整和变化。

例如网站的登录表单作为一个关键功能，往往在多个测试用例中出现。由于登录会涉

及到用户名、密码的文本框和提交按钮，缺乏良好的代码组织结构会导致大量重复的代码查找、填写文本框操作。如果登录元素的文档结构发生了变化，则重构测试脚本的查找逻辑会带来更多的维护工作量，甚至可能因为修改不完全导致无谓的脚本运行错误，需要额外的人力进行排错。因此，如何编写复用率高、维护成本低的脚本成为一个自动化测试项目能否成功的关键。

13.1.1　关注点分离

页面对象模型（Page Object[①]）是自动化测试中至关重要的设计模式，核心是基于关注点分离（Separation of Concerns）的思想，对页面元素和方法实现最大程度的封装与复用。关注点分离于1974年由Edsger W. Dijkstra提出[②]，是处理复杂性问题的方法论。关注点混杂在一起会导致复杂性的大大增加，把关注点分离出来，进行标示、封装和操纵可以化繁为简，让复杂问题简单化。关注点分离的思想在面向对象的程序设计领域得到了蓬勃发展，特定领域的实现会从业务逻辑流中独立出来，封装成类或函数，这样原来分散在整个应用程序中的变动就可以很好的管理起来。

页面对象模型正是基于关注点分离思想把页面中的公用对象和公用方法封装起来的设计模式。图13-1所示是Martin Fowler对页面对象模型的概括。在页面对象模型中，公用对象包括被重复使用的文本框、按钮等，被封装后在编写脚本时可随时调用。当这些对象的属性因为需求变更而需要修改时，只需修改该对象的封装部分即可，而无需修改所有的相关测试脚本。公用方法包括页面中的各独立功能，封装后可被直接调用而无需关注其内部实现。

基于关注点分离的思想，自动化测试框架往往由两层组成，基于具体项目的需求，可以由相同或不同的开发人员分别编写完成。底层为封装后的页面对象，可以供测试脚本直接调用；上层为具体的测试用例，调用底层的页面对象，组装为一个复杂的工作流。在实际实施过程中，上层的测试用例往往不用关心具体页面对象的内部实现逻辑，只需要负责进行业务组装即可；同样的，如果某个具体功能点的内部实现发生了变化，只需要修改对象模型而不用重构上层脚本，这也是其可以由不同的开发人员协同工作完成的基础。

---

① Martin Fowler. PageObject[OL]. 2013. http://martinfowler.com/bliki/PageObject.html.
② Edsger W. Dijkstra. On the role of scientific thought[OL]. 1974. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html/.

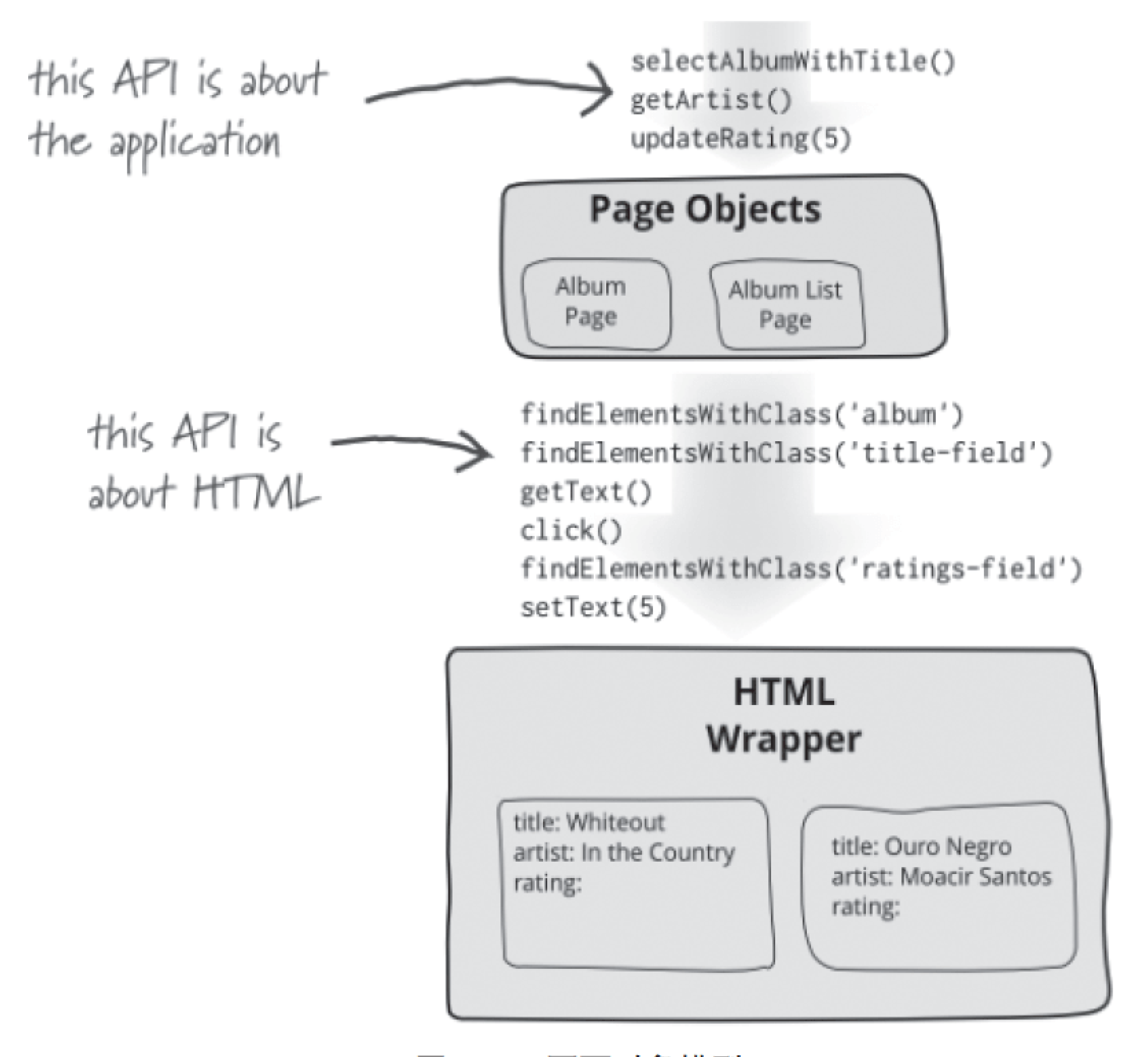


图13-1　页面对象模型

13.1.2　实现Protractor页面对象

实现页面对象的关键是以面向对象的思想对页面进行封装，每个页面以一个类的形式存在。

对于在Java和C#中如何定义一个类，大家都比较熟悉，但在JavaScript中该如何用类定义页面对象呢？

本节将以http://angular.github.io/angular-phonecat/step-14/app作为被测对象，演示在Protractor中实现页面对象的方法。这是一个用于AngularJS教学的著名示例，包括有大量AngularJS的基础知识和实现方法。

该示例包含两个页面，主页以列表的形式显示各种品牌的移动设备，同时支持关键字查找和排序，如图13-2所示。

在该主页中单击任何一个移动设备即进入设备详情页面，包括各缩略图和技术指标，如图13-3所示。



图13-2　被测应用主页

图13-3　被测应用详细页面

首先搭建Protractor测试环境以及下载相关的浏览器驱动。创建本地文件夹phonecat-e2e，启动命令控制台后执行以下命令：

```
npm init

npm install protractor --save-dev

node .\node_modules\protractor\node_modules\webdriver-manager update --ie --ie32
```

以下为命令执行后的package.json文件，可以看到目前的测试仅对protractor包有依赖。

```
{
  "name": "phonecat-e2e",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "protractor": "^4.0.11"
  }
}
```

在phonecat-e2e文件内创建两个子文件夹page-objects和specs，分别用于存放封装好的页面对象和测试用例。请注意，页面对象是与测试无关的独立个体，可以供多个测试用例共享，在实际操作中，页面对象与测试用例经常由不同的开发人员完成。

接下来创建protractor.local.conf.js并配置如下。为了让配置代码尽量精简，方便阅读，当前仅使用了基本的配置，包括指定Chrome用于测试，指定测试用例文件夹为specs，指定单元测试框架为Jasmine等。

```
exports.config = {
  directConnect: true,
  specs: [
    'specs/*.js'
  ],
  capabilities: {
    'browserName': 'chrome'
  },
  baseUrl: 'http://angular.github.io/',
```

```
  framework: 'jasmine2',

  jasmineNodeOpts: {

    defaultTimeoutInterval: 30000

  }

};
```

图13-4为当前设置好的目录结构。

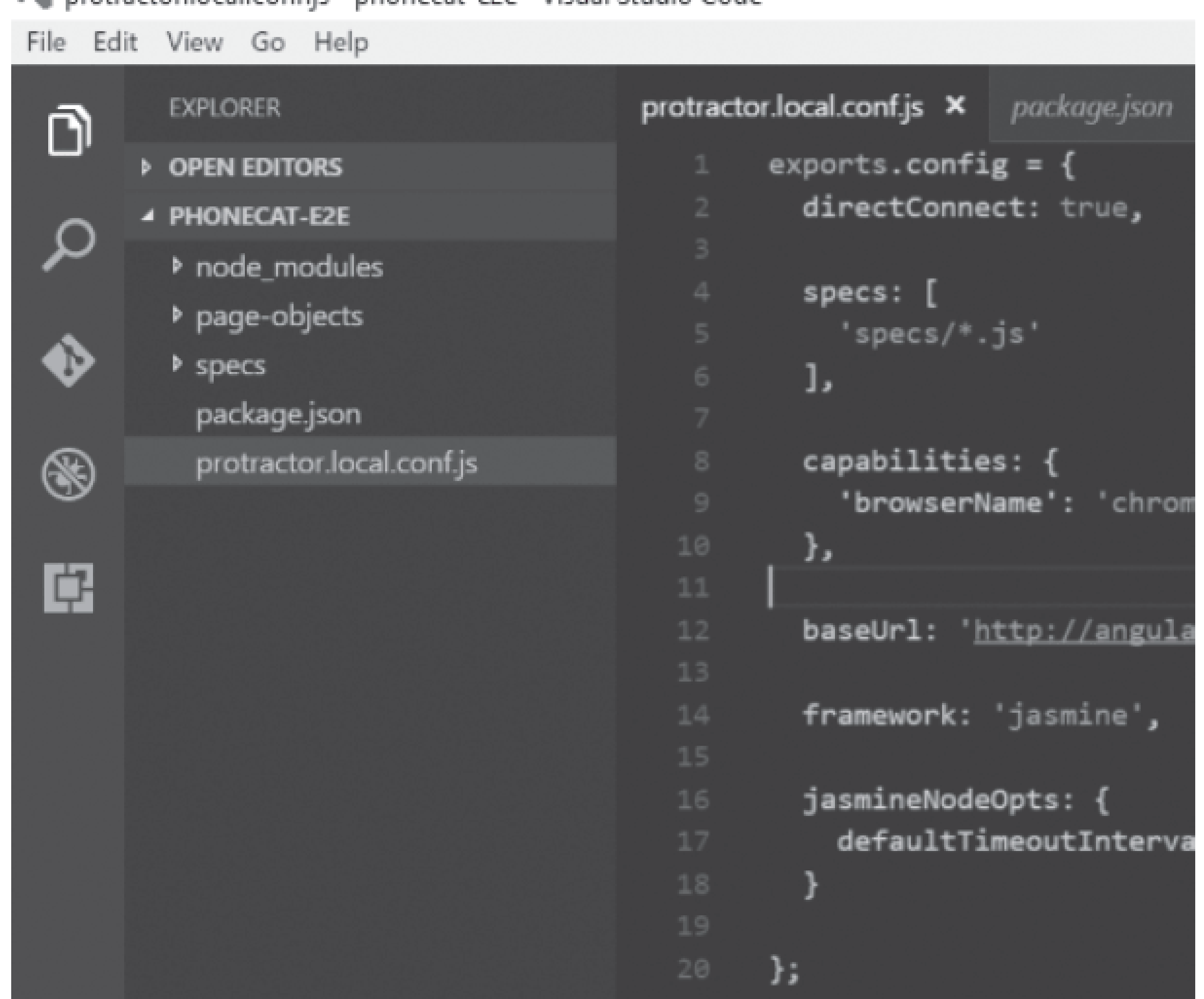


图13-4　文件结构

接下来开始写页面对象。面向对象编程的一个重要功能是通过继承，让子类无需重复编写代码即可复用基类定义好的属性和方法。作为页面，它们同样共享一些相同的功能，例如返回到Home页面，返回当前的页面地址等。但是，JavaScript中只有对象并没有类的概念，该如何实现类似的继承功能用于定义页面对象的基类呢？

JavaScript是一种基于原型的语言，与传统的Java和C#不同，它的核心理念是使用原型对象作为模板来创建新的对象。被创建的新对象不但拥有自己创建时定义的属性，还可以享有原型对象的属性。在这里，可以把页面对象的基类理解为模板对象，基于基类模板对象创建出的页面对象可以直接访问基类中的属性。

在page-objects文件夹内创建base-page.js，它是模板基类对象，实现了页面的共享属性和方法，代码如下：

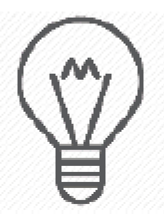
```
function BasePage(){

}
```

```
BasePage.prototype = Object.create({}, {
    absoluteUrl: {get: function () {return browser.getLocationAbsUrl();}},
    goHome: {value: function () {browser.get('angular-phonecat/step-14/app');}}
});
module.exports = BasePage;
```

以上代码中通过调用Object.create创建了一个新的对象，该对象包含一个goHome方法用于返回主页面，以及属性absoluteUrl用于获得当前的页面地址。Object.create提出于ECMAScript 5.1，可以通过第1个参数指定新对象的模板对象，在本代码中，BasePage的模板对象为一个空对象。

那么最后一句又是什么作用呢？别忘了，Protractor的测试脚本是运行在Node.js环境中的。Node.js通过实现CommonJS的Modules/1.0标准引入了模块概念，一个模块可以通过module.exports将函数或变量导出。在以上代码中，BasePage被导出后可以被其他脚本通过require函数引入。



Node.js的模块加载于2013年脱离了CommonJS独立发展[①]，但仍然遵循相似的语法。

接下来新建phone-list-page.js文件，它包含类PhoneListPage的实现如下：

```
var BasePage = require('./base-page.js');
function PhoneListPage ()
{
    this.phoneList = element.all(by.repeater('phone in $ctrl.phones'));
    this.query = element(by.model('$ctrl.query'));
    this.queryField = element(by.model('$ctrl.query'));
    this.orderSelect = element(by.model('$ctrl.orderProp'));
    this.nameOption = this.orderSelect.element(by.css('option[value="name"]'));
    this.phoneNameColumn = element.all(by.repeater('phone in $ctrl.phones').column('phone.
name'));
    this.links = element.all(by.css('.phones li a'));
```

① Node.js. Breaking the CommonJS standardization impasse[OL]. 2013. https://github.com/nodejs/node-v0.x-archive/issues/5132#issuecomment-15432598.

```
    this.imagelinks = element.all(by.css('.phones li a img'));
}
PhoneListPage.prototype = Object.create(BasePage.prototype, {
    phonesCount: {get: function () {
        return this.phoneList.count();
    }},
    phoneListNames: {get: function () {
        return this.phoneNameColumn.map(function(elem) {
          return elem.getText();
        });
    }},
    linksCount: {get: function () {
        return this.links.count();
    }},
    imageLinksCount: {get: function () {
        return this.imagelinks.count();
    }},
    clickLink: {value: function (idx) {
        return this.links.get(idx).click();
    }},
    clickImageLink: {value: function (idx) {
        return this.imagelinks.get(idx).click();
    }},
    clearQueryField: {value: function () {
        this.queryField.clear();
    }},
    setQueryField: {value: function (query) {
        this.queryField.sendKeys(query);
    }},
    orderByName: {value: function () {
```

```
      this.nameOption.click();
    }}
});
module.exports = new PhoneListPage();
```

以上代码首先在函数PhoneListPage中，通过Protractor元素定位器声明了页面中各个元素，包括手机设备列表、搜索关键字文本框和排序下拉框等。声明了这些页面元素后，页面对象的其他部分代码只需直接引入即可，不需要重复使用元素定位器。如果后期需求变更发生了元素定位的变化，也只需要修改PhoneListPage函数即可，大大降低了脚本的维护难度。

作为页面对象，Object.create被再次使用，这次BasePage的原型对象被作为参数传入Object.create中，也意味着PhoneListPage的基类是BasePage，它可以直接访问BasePage中定义的方法与属性。同时，PhoneListPage页面对象本身也被赋予了该页面独有的属性和方法，例如phonesCount用于获得当前的手机设备数量，setQueryField用于设置搜索文本框等。

读者可能会奇怪，既然所有被使用的元素都已经通过定位器在PhoneListPage函数中做了声明，那么在测试脚本里直接通过声明好的元素变量进行操作就行了，为什么还要再次封装函数呢？别忘了关注点分离的思想。页面对象模型的概念不仅仅在于封装，也要求测试脚本并不需要了解页面内的技术细节就可以基于页面对象完成测试用例，只有这样，页面对象和测试用例才能充分解耦，让一部分开发人员能够专注于开发页面对象，另一部分开发人员无需深刻了解Protractor元素定位器也能进行测试用例的开发。而要满足以上这些，必要条件就是页面里的所有功能都要通过属性或方法进行暴露，测试脚本无需直接访问页面元素。

代码的最后一行，是通过module.exports把创建好的PhoneListPage对象导出，供测试脚本使用，也就是接下来要创建的phone-list.js。由于它是测试用例，本示例将其放置在specs文件夹下。以下为具体的代码：

```
var listpage = require('../page-objects/phone-list-page.js');
describe('View: Phone list', function() {
    beforeEach(function() {
      listpage.goHome();
    });
    it('should redirect to /phones', function() {
```

```
      expect(listpage.absoluteUrl).toBe('/phones');
    });
    it('should filter the phone list as a user types into the search box', function() {
      expect(listpage.phonesCount).toBe(20);
      listpage.setQueryField('Dell');
      expect(listpage.phonesCount).toBe(2);
      listpage.clearQueryField();
      expect(listpage.phonesCount).toBe(20);
    });
    it('should be possible to control phone order via the drop-down menu', function() {
      listpage.setQueryField('4G');
      expect(listpage.phoneListNames).toEqual([
        'MOTOROLA ATRIX\u2122 4G',
        'T-Mobile myTouch 4G',
        'T-Mobile G2'
      ]);
      listpage.orderByName();
      expect(listpage.phoneListNames).toEqual([
        'MOTOROLA ATRIX\u2122 4G',
        'T-Mobile G2',
        'T-Mobile myTouch 4G'
      ]);
    });
    it('should render phone specific links', function() {
      listpage.setQueryField('LG');
      expect(listpage.linksCount).toBe(2);
      listpage.clickLink(0);
      expect(listpage.absoluteUrl).toBe('/phones/lg-axis');
    });
});
```

可以看到，基于页面对象开发的测试代码简单易懂，无需具体了解页面内的实现，即可通过调用页面对象编写脚本。例如在以下测试用例中，可以很容易地理解：该用例首先判断出页面中的手机设备个数为20，而当以Dell作为关键字进行搜索时，则只应该搜索到2个手机设备；最后清除搜索关键字，再次返回20个手机设备。在这里，开发人员既不用关心如何找到设备列表元素，也不需要关心如何找到搜索关键字元素，因此可以把更多的精力集中到具体的业务逻辑中。

```
it('should filter the phone list as a user types into the search box', function() {
  expect(listpage.phonesCount).toBe(20);
  listpage.setQueryField('Dell');
  expect(listpage.phonesCount).toBe(2);
  listpage.clearQueryField();
  expect(listpage.phonesCount).toBe(20);
});
```

在该网站中，单击任何一个手机设备会进入该设备的具体页面，以下分别为对应的页面对象和测试用例的代码。

页面对象：page-objects\phone-detail-page.js

```
var BasePage = require('./base-page.js');
function PhoneDetailPage ()
{
   this.phoneName = element(by.binding('$ctrl.phone.name'));
   this.mainImage = element(by.css('img.phone.selected'));
   this.thumbnails = element.all(by.css('.phone-thumbs img'));
}
PhoneDetailPage.prototype = Object.create(BasePage.prototype, {
   phoneNameText: {get: function () {return this.phoneName.getText();}},
   mainImageSrc: {get: function () {return this.mainImage.getAttribute('src');}},
   clickThumbnail: {value: function (idx) {this.thumbnails.get(idx).click();}}
});
```

```
module.exports = new PhoneDetailPage();
```

测试用例：specs\phone-detail.js

```
var detailpage = require('../page-objects/phone-detail-page.js');
var listpage = require('../page-objects/phone-list-page.js');
describe('View: Phone detail', function() {
  beforeEach(function() {
    listpage.goHome();
    listpage.setQueryField('nexus-s');
    listpage.clickLink(0);
  });
  it('should display the `nexus-s` page', function() {
    expect(detailpage.phoneNameText).toBe('Nexus S');
  });
  it('should display the first phone image as the main phone image', function() {
    expect(detailpage.mainImageSrc).toMatch(/img\/phones\/nexus-s.0.jpg/);
  });
  it('should swap the main image when clicking on a thumbnail image', function() {
    detailpage.clickThumbnail(2);
    expect(detailpage.mainImageSrc).toMatch(/img\/phones\/nexus-s.2.jpg/);
    detailpage.clickThumbnail(0);
    expect(detailpage.mainImageSrc).toMatch(/img\/phones\/nexus-s.0.jpg/);
  });
});
```

考虑到所用技巧与之前主页类似，作者不再对以上代码逐一解释。请读者注意的是，由于需要先访问手机列表页面，选择了手机型号后再进入具体设备页面，因此该测试用例实际上对两个页面对象都有依赖，这也是为什么在该测试用例的首行同时引用了phone-detail-page.js和phone-list-page.js的原因。对于实际生产环境中较复杂的业务逻辑，多页面跳转是很常见的情况，建议读者参考以上代码进行实施。

13.1.3 页面对象最佳实践

页面对象模型的实现是一个复杂的过程，对开发人员的编程能力要求较高。读者在实施过程中，建议基于以下最佳实践构建自己的Protractor自动化测试框架：

1. 一个页面对应一个或多个页面对象

一个页面至少由一个页面对象与其对应，一对一的抽象关系方便测试人员能够在实际页面和其代码实现之间迅速切换。对于页面中包含的复杂元素，或者某些复合控件被多个页面引用的情况，建议将复杂元素抽象为一个独立的页面对象。

2. 一个页面对象对应一个文件

正如C#中常把一个类封装到一个.cs文件中一样，每一个页面对象应该实现到各自独立的.js文件中，从而保持代码结构的整齐。反之，如果一个文件中有多个页面对象，测试人员需要花费更多的精力用于搜索、比对代码，从而大大增加维护费用。

3. 在文件首行引用其他页面对象

复杂的页面对象或测试用例往往会引用到多个页面对象。有些开发人员习惯于按实际的业务逻辑，在代码执行过程中，当需要某个页面对象的时候再添加引用。尽管在技术层面上这样做没有问题，但这种按需引用的方式无法直观地体现页面与页面，或者测试用例与页面之间的依赖关系。在文件首行即添加所有页面对象的引用，让依赖关系一目了然，可以有效提高代码的可读性，降低开发人员的学习难度。

4. 调用页面对象暴露的方法或属性而不是元素

正如上一节示例代码所示，测试框架中底层的页面对象和上层的测试用例往往由不同的开发人员完成。高效的敏捷开发要求开发人员关注业务逻辑而无需关心页面内每个元素的具体类型和支持的属性和方法。为了满足这个要求，页面里的所有功能都应该通过属性或方法进行暴露，而测试脚本无需直接访问页面元素。

5. 避免在页面对象中使用断言

页面对象是对业务逻辑的抽象和封装，是对页面操作的代码体现，而这与测试无关。如果将测试用例与页面对象混为一谈，在页面对象中进行断言，则不仅会让页面对象的实现变得更复杂失去纯粹性，测试用例也会因为分散在多个文件中的断言降低了可读性和可维护性。

6. 合理的文件结构

保持合理的文件结构的目的是可读性，那什么是合理的文件结构呢？它应该符合DRY

（Don't Repeat Yourself[①]）原则，即保证开发人员在开发和维护过程中，可以迅速找到目标文件，而不需要每次打开项目后再次寻找。原始的代码文件、单元测试文件和自动化测试文件可以共同存在于一个项目内，但它们之间需要保持独立。测试文件的结构要与原始代码文件的结构保持一致，从而根据相互的映射关系迅速定位文件。例如下面的文件结构。

```
| -- app
  | -- css
  | -- img
  | -- js
  | -- html
     home.js
     profile.js
     contacts.html
 | -- unittest
 | -- e2e
  | -- page-objects
    | -- home-page.js
    | -- profile-page.js
    | -- contacts-page.js
  | -- specs
    | -- home-spec.js
    | -- profile-spec.js
    | -- contacts-spec.js
```

13.2　数据驱动测试

在测试领域，数据驱动测试是指测试用例里操作的数据不是内嵌在测试用例里，而

① Bill Venners. Orthogonality and the DRY Principle[OL]. 2003. http://www.artima.com/intv/dry.html.

是由外部数据源提供。数据驱动测试的概念并不为自动化测试所独有，它同样适用于单元测试。同时，它与具体使用什么测试框架无关，无论是Java的TestNG还是JavaScript里的Jasmine或Cucumber，只要从事测试就有数据驱动的需求以及不同的实施方法。

为什么要把测试数据放到外部数据源，这样不是反而降低了测试用例的可读性吗？设想当前需要测试某网站的新用户注册流程。一个符合设计要求的注册模块，会对用户名进行校验，例如用户名内需要同时包含大小写英文字母，需要包含某些允许的特殊字符，用户名和密码不能相似等。在写测试用例的时候，测试人员会发现以上3个校验功能存在相互影响因素，为了避免干涉，需要用多个不同的测试用例才能全面覆盖这些校验点。而另一方面，这些测试用例的操作步骤却是一样的，都是输入用户名和密码，单击注册按钮，唯一不同的就是用于测试的用户名和密码不同。测试人员当然可以分开实现这些测试用例，但重复代码出现了，这不符合DRY原则。而且，如果该注册模块的功能继续加强，就只能继续添加更多的测试用例。

为了解决以上痛点，数据驱动测试理论出现了，它的核心思想是数据和测试代码分离，测试用例只规定业务逻辑，所有的输入和检验值由外部数据源定义，这样设计并不会影响执行结果，但可以重用测试用例。基于数据驱动搭建的自动化测试框架有以下优点：

1. 充分共享数据源

同样的测试数据可以被多个测试用例使用，提高数据利用率。

2. 可重复性

在保持测试用例不变的情况下，可以用不同的边界数据执行多次测试，提高测试的有效性。

3. 数据与测试代码分离

基于数据驱动的测试让数据与测试代码解耦，让开发人员能够将主要精力花费在业务流程和测试用例本身，而不需要考虑到各种输入的全部可能性，提高代码效率。

4. 灵活添加数据

既然开发人员的精力在测试用例本身，那谁来提供这么多的外部数据呢？由于数据已经与测试代码分离，任何测试人员发现了一个可能没有被覆盖的边界条件时，都可以随时方便地更新数据源而不用担心其是否会影响业务逻辑。这种简化的工作流程保证了更多的有效测试数据，对测试质量影响深远。

根据实际情况下的业务各有不同，数据源有多种选择，包括.xls、.xlsx、.csv文件或者数据库等。在TestNG中可以使用DataProvider声明资源内嵌型数据，或者通过读文件从.csv获得数据。

本书的Protractor自动化测试是基于Jasmine单元测试框架搭建的，是否有对应的读取外部数据源的方式呢？当然有，不过，由于Protractor是运行在Node.js环境中，使用任何.xls、.csv文件或数据库的外部数据源都是可行的，但在JavaScript环境下，最简便也是使用最广泛的外部数据源是JSON文件，基于JSON的数据无需任何显式的文件读取或连接即可直接使用。

现在回顾一下13.1.2节phone-list.js中的如下测试用例：

```
it('should be possible to control phone order via the drop-down menu', function() {
     listpage.setQueryField('4G');
     expect(listpage.phoneListNames).toEqual([
       'MOTOROLA ATRIX\u2122 4G',
       'T-Mobile myTouch 4G',
       'T-Mobile G2'
     ]);
     listpage.orderByName();
     expect(listpage.phoneListNames).toEqual([
       'MOTOROLA ATRIX\u2122 4G',
       'T-Mobile G2',
       'T-Mobile myTouch 4G'
     ]);
   });
```

该测试用例在手机设备列表页面使用关键字搜索后，先对搜索结果进行检查，然后把搜索结果按名字顺序重新排序并再次检查排序结果是否符合预期。该测试用例本身逻辑较为简单，但因为测试数据比较多，反而影响了测试用例的阅读。如果基于数据驱动，可以按如下方法加以改进。

（1）在命令控制台执行以下命令安装jasmine-data-provider[①]，这是一个Jasmine的插件，可以把数据通过回调函数传递给测试用例。

```
npm install jasmine-data-provider -save-dev
```

① Mortal Flesh. Jasmine-data-provider[OL]. [2016]. https://github.com/MortalFlesh/jasmine-data-provider.

（2）新建文件夹testdata，创建外部数据phone-list-data.json。在以下示例代码包括两组测试数据，分别以4G和Samsung作为filter对设备进行搜索。

```
{
   "controlphoneorder": [
      {
         "filter": "4G",
         "sortbyage": [
            "MOTOROLA ATRIX\u2122 4G",
            "T-Mobile myTouch 4G",
            "T-Mobile G2"
         ],
         "sortbyname": [
            "MOTOROLA ATRIX\u2122 4G",
            "T-Mobile G2",
            "T-Mobile myTouch 4G"
         ]
      },
      {
         "filter": "Samsung",
         "sortbyage": [
            "Samsung Gem\u2122",
            "Samsung Galaxy Tab\u2122",
            "Samsung Showcase\u2122 a Galaxy S\u2122 phone",
            "Samsung Mesmerize\u2122 a Galaxy S\u2122 phone",
            "Samsung Transform\u2122"
         ],
         "sortbyname": [
            "Samsung Galaxy Tab\u2122",
            "Samsung Gem\u2122",
```

```
            "Samsung Mesmerize\u2122 a Galaxy S\u2122 phone",
            "Samsung Showcase\u2122 a Galaxy S\u2122 phone",
            "Samsung Transform\u2122"
        ]
    }
  ]
}
```

（3）在specs\phone-list.js文件里分别添加对jasmine-data-provider和phone-list-data.json的引用，代码如下：

```
var using = require('jasmine-data-provider');
var testdata = require('../testdata/phone-list-data.json');
```

（4）修改测试用例如下，使得通过using将外部数据作为数组传入jasmine-data-provider后，其内部实现会遍历每一组数据并分别执行测试。与未修改前的测试代码相比，以数据驱动的测试用例更整洁，可读性更好。

```
using(testdata.controlphoneorder, function(inputdata){
    it('should be possible to control phone order via the drop-down menu', function() {
      listpage.setQueryField(inputdata.filter);
      expect(listpage.phoneListNames).toEqual(inputdata.sortbyage);
      listpage.orderByName();
      expect(listpage.phoneListNames).toEqual(inputdata.sortbyname);
    });
  });
```

13.3　测试报告

在自动化测试中，信息丰富和多样化的测试报告是非常重要的一环。简言之，测试报

告就是把测试的过程和结果生成文档，既可以传递给持续的集成环境，也可以供开发测试人员复查；对发现的问题和缺陷进行分析。测试报告为纠正软件的质量问题提供了坚实的数据依据，同时能够为软件验收和交付打下良好基础，是测试行为中不可或缺的一部分。Protractor本身只生成基本的测试报告，缺乏可定制性，需要使用插件满足多样化的测试报告需求。

13.3.1 控制台报告

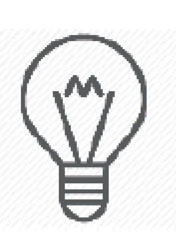
通过jasmine-spec-reporter①插件，可以在控制台内生成丰富的测试报告，基于不同的颜色和符号标注出不同的测试结果，适合测试人员在脚本开发阶段，实时通过控制台报告检查测试的运行情况。

（1）在命令控制台执行以下命令安装插件。

```
npm install jasmine-spec-reporter --save-dev
```

（2）修改protractor配置文件，添加以下代码指定生成控制台报告。

```
onPrepare: function() {
   var SpecReporter = require('jasmine-spec-reporter');
   jasmine.getEnv().addReporter(new SpecReporter());
  }
```



onPrepare在执行测试用例之前被调用，可以用于定制和初始化测试环境，这是Protractor中常用的技巧。类似的回调函数还有OnComplete和onCleanUp，分别在测试用例执行结束以及WebDriver对象关闭后被调用。

（3）运行测试用例，以下为生成的控制台报告，默认成功执行的测试用例会以绿色及对勾符号（√）进行标注。控制台报告有良好的定制性，请参考网址https://github.com/bcaudan/jasmine-spec-reporter修改默认配置。

① Bastien Caudan. Jasmine-spec-reporter[OL]. [2016]. https://github.com/bcaudan/jasmine-spec-reporter.

```
[11:02:43] I/direct - Using ChromeDriver directly...
[11:02:43] I/launcher - Running 1 instances of WebDriver
Started
Spec started

  View: Phone detail
    √ should display the `nexus-s` page
     √ should display the first phone image as the main phone image
     √ should swap the main image when clicking on a thumbnail image

   View: Phone list
    √ should redirect to /phones
     √ should filter the phone list as a user types into the search box
     √ should be possible to control phone order via the drop-down menu
     √ should render phone specific links



7 specs, 0 failures
Finished in 23.323 seconds

Executed 7 of 7 specs SUCCESS in 23 secs.
```

图13-5　控制台报告

13.3.2　JUnit报告

JUnit[①]是一种基于XML的报告格式，被广泛应用于各持续集成系统内。可以通过插件jasmine-reporters[②]为Protractor添加JUnit报告功能，同时该插件还支持NUnit等其他报告格式。

（1）在命令控制台执行以下命令安装插件。

```
npm install jasmine-reporters --save-dev
```

（2）修改protractor配置文件，添加以下代码指定生成JUnit测试报告，savePath用于指定报告的保存位置。

```
onPrepare: function() {
  var jasmineReporters = require('jasmine-reporters');
  jasmine.getEnv().addReporter(new jasmineReporters.JUnitXmlReporter({
      consolidateAll: true,
      savePath: 'testresults',
```

① Windy Road Technology. JUnit-Schema[OL]. [2016]. https://github.com/windyroad/JUnit-Schema/blob/master/JUnit.xsd.

② Larry Myers. Jasmine-reporters[OL]. [2016]. https://github.com/larrymyers/jasmine-reporters.

```
        filePrefix: 'xmloutput'
    }));
}
```

（3）运行测试用例。以下为生成的示例报告。

```
<?xml version="1.0" encoding="UTF-8" ?>
<testsuites>
 <testsuite name="View: Phone detail" timestamp="2016-11-28T11:55:15" hostname="localhost"
time="14.344" errors="0" tests="3" skipped="0" disabled="0" failures="0">
 <testcase classname="View: Phone detail" name="should display the `nexus-s` page"
time="6.851" />
 <testcase classname="View: Phone detail" name="should display the first phone image as
the main phone image" time="3.704" />
 <testcase classname="View: Phone detail" name="should swap the main image when clicking
on a thumbnail image" time="3.788" />
 </testsuite>
 <testsuite name="View: Phone list" timestamp="2016-11-28T11:55:29" hostname="localhost"
time="9.492" errors="0" tests="4" skipped="0" disabled="0" failures="0">
 <testcase classname="View: Phone list" name="should redirect to /phones" time="1.014" />
 <testcase classname="View: Phone list" name="should filter the phone list as a user types
into the search box" time="2.53" />
 <testcase classname="View: Phone list" name="should be possible to control phone order
via the drop-down menu" time="2.499" />
 <testcase classname="View: Phone list" name="should render phone specific links"
time="3.448" />
 </testsuite>
</testsuites>
```

jasmine-reporters功能强大，关于它的其他配置属性，请参考网址https://github.com/larrymyers/jasmine-reporters中所述。本书将在第16章介绍如何把JUnit测试报告集成到持续

集成环境内。

13.3.3　HTML报告

基于XML的JUnit报告适合于持续集成系统，但缺乏多样化的颜色与格式对测试结果进行区分与标注，所以不适合直接阅读。为了提供用户更好的报告体验形式，可以通过插件protractor-jasmine2-html-reporter[①]为Protractor添加HTML报告功能。

（1）在命令控制台执行以下命令安装插件。

```
npm install protractor-jasmine2-html-reporter --save-dev
```

（2）修改Protractor配置文件，添加以下代码指定生成HTML测试报告，savePath用于指定HTML文件的保存路径。该插件同时支持为每个测试用例或失败的用例生成截图，screenshotsFolder属性用于指定截图的保存路径。

```
onPrepare: function() {
  var jasmine2HtmlReporter = require('protractor-jasmine2-html-reporter');
  jasmine.getEnv().addReporter(
    new jasmine2HtmlReporter({
      savePath: './htmlreports/',
      screenshotsFolder: 'images',
      takeScreenshots: true,
      cleanDestination: true
    })
  );
}
```

（3）运行测试用例。以下为生成的示例报告。关于protractor-jasmine2-html-reporter的其他配置属性，请参考网址https://github.com/Kenzitron/protractor-jasmine2-html-reporter中所述。本书将在第16章介绍如何将HTML报告集成到持续集成环境的方法。

① Kenzitron. protractor-jasmine2-html-reporter[OL]. [2016]. https://github.com/Kenzitron/protractor-jasmine2-html-reporter.

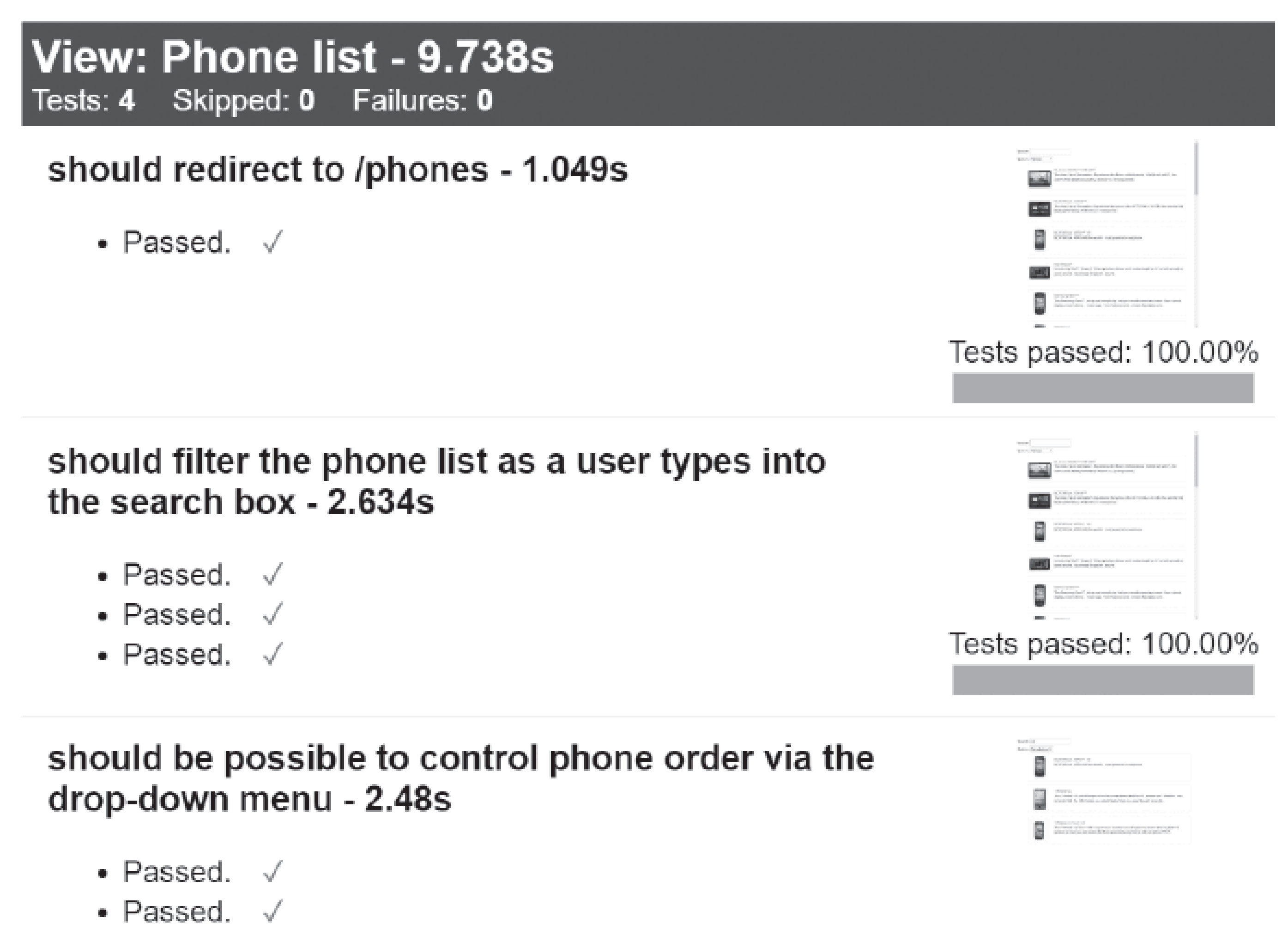


图13-6 HTML报告

13.4 性能测试

过去，对于软件服务，各厂商首先关心的往往是软件实现的功能多不多，却容易忽略软件性能对产品的影响。

近几年，软件服务特别是Web应用在性能方面得到了越来越多的重视，各厂商意识到在功能同质化的情况下，更优越的性能已经成为客户选择的重要指标。Stack Overflow的联合创始人Jeff Atwood曾提出Performance is a Feature的论点，把性能指标提高到了与功能一样重要的层次[①]。

当前，性能测试已经成为产品研发中必不可少的一环。通过性能测试可以：

- 了解产品的性能情况，检验产品性能是否满足业务需求。
- 量化并找出产品的性能瓶颈，为进一步优化提供数据。
- 量化性能指标，为销售人员提供性能方面的建议。

性能测试本身是一个信息收集和分析的过程，它与当前测试环境以及操作速度有很大关系，如果使用手工测试往往无法保证操作的协调性，因而难以收集到可靠的性能测试结果。在基于Protractor的自动化测试中，结合插件protractor-perf[②]则可以自动收集浏览器的

---

① Jeff Atwood. Performance is a Feature[OL]. 2011. https://blog.codinghorror.com/performance-is-a-feature/.

② Parashuram N. protractor-perf[OL]. [2016]. https://github.com/axemclion/protractor-perf.

性能指标，避免人工测试中的种种弊端。

（1）插件protractor-perf依赖于某些原生模块，需要node-gyp进行编译[①]，环境要求包括：

- 安装Visual C++ Build Tools或者Visual Studio 2015。
- 从网址https://www.python.org/downloads/处下载并安装Python 2.7（当前最新的node-gyp不支持v3.x.x版本的Python）。

（2）在命令控制台执行以下命令安装protractor-perf插件。

```
npm install protractor-perf -save-dev
npm install protractor-perf -g
```

（3）在测试用例中通过PerfRunner对象对性能测试的启动和结束进行控制，调用getStats并传入对应的性能指标名称获得收集到的性能数据。

以下为specs/perf/phone-list.js文件的示例代码：

```
var listpage = require('../../page-objects/phone-list-page.js');
var PerfRunner = require('protractor-perf');
describe('Performance: Phone list', function() {
    var perfRunner = new PerfRunner(protractor, browser);
    beforeEach(function() {
      listpage.goHome();
    });
    it('meanFrametime in query', function() {
      perfRunner.start();
      listpage.setQueryField('LG');
      perfRunner.stop();
      //perfRunner.printStats();
      // call this function to get the baseline
      expect(perfRunner.getStats('meanFrameTime')).toBeLessThan(20);
    });
});
```

① Node.js. node-gyp[OL]. [2016]. https://github.com/nodejs/node-gyp.

插件protractor-perf基于browser-perf[①]所构建，可以记录浏览器的时间轴指标以及事件触发的事件点，如图13-7所示。请参考网址https://github.com/axemclion/browser-perf/wiki/Metrics以获取所有的性能指标说明。

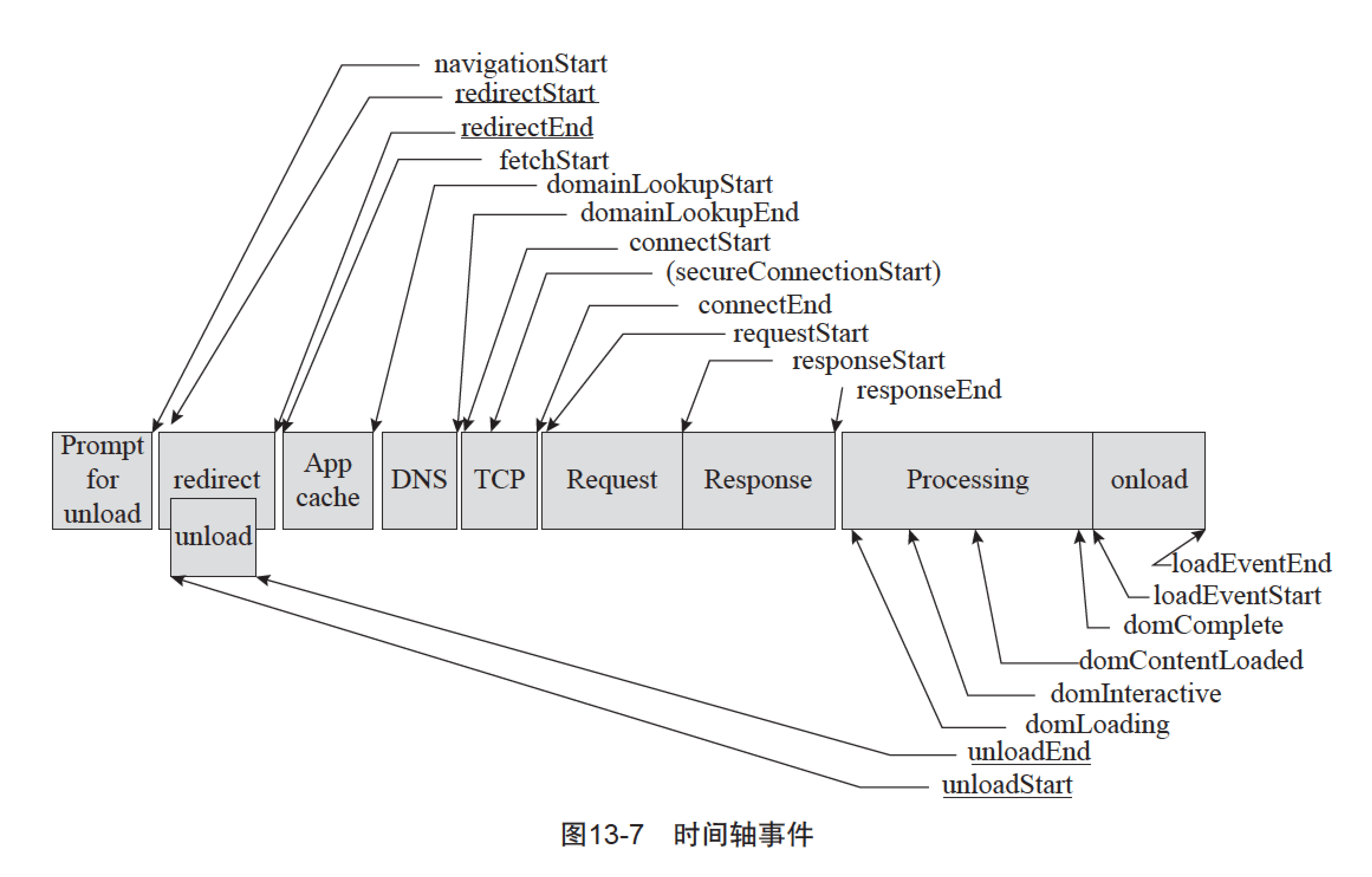


图13-7　时间轴事件

（4）以下为性能测试配置文件protractor-perf.js的示例代码，与Protractor配置文件格式和所支持的参数一致。

单独设置一个性能测试配置文件的原因是，性能测试往往通过处理较大的数据量来暴露产品的性能问题，而这在普通的页面测试中不是必须的，甚至会影响到普通页面测试的完成效率。所以，在性能测试最佳实践中，建议同一个测试用例里不要同时覆盖普通的页面测试和性能测试，而是建立专门的性能测试用例。在以下配置文件中，对应的性能测试用例集合被命名为perf，定义在配置属性suites中。

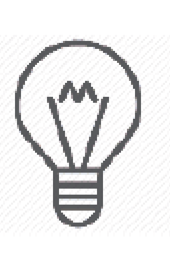
```
exports.config = {
  selenium: 'http://localhost:4445/wd/hub',
  seleniumPort: 4445, // Port matches the port above
  suites:{
```

① Parashuram N. browser-perf[OL]. [2016]. https://github.com/axemclion/browser-perf.

```
    perf: 'specs/perf/*.js'
  },
  capabilities: {
    'browserName': 'chrome'
  },
  baseUrl: 'http://angular.github.io/',
  framework: 'jasmine2',
  jasmineNodeOpts: {
    defaultTimeoutInterval: 30000
  }
};
```

（5）在命令控制台调用以下命令执行性能测试，参数是性能测试集合所对应的名字。

```
protractor-perf perf.conf.js --suite perf
```



如果在测试中出现错误信息“ConfigParser is not a constructor error”，请根据网址https://github.com/afterbangx/protractor-perf/commit/e58d478abc8eb641ac06b1745d73746cb561bfd0中的信息修改lib/cli.js文件。

13.5　图像匹配

基于Protractor提供的元素定位器，可以很容易地检查出页面元素是否存在、数量是否正确，以及DOM结构是否符合设计等问题。但如果要检查页面的布局，包括图片是否发生了拉伸、文本框之间是否发生了重叠却难度较大。

特别是当前Web前端流行的是响应式设计，一个成熟的网站往往能够根据设备环境（操作系统、屏幕尺寸和分辨率等）自动调整页面布局。针对这种情况，如果在脚本里进行布局检查，获取每个元素的位置及之间的层叠关系会使脚本变得复杂而难以维护，这不仅需花费大量的人力物力，而且最后也很难成功。

针对这样的需求，推荐的方式是对页面截屏，基于基准图片进行实时的图像匹配。在

Protractor框架下可以通过插件pix-diff[①]实现。pix-diff依赖于另一个模块blink-diff[②]构建，进行图像比较，具体步骤如下。

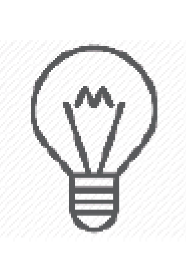
（1）在命令控制台执行以下命令安装插件。

```
npm install -save-dev pix-diff
```

（2）修改protractor配置文件，添加以下代码启用并设置插件。

```
onPrepare: function() {
    var PixDiff = require('pix-diff');
    browser.pixDiff = new PixDiff({
        basePath: './screenshots/',
        baseline: true,
        diffPath: './screenshots/',
        formatImageName: '{tag}-{browserName}-{width}x{height}-dpr-{dpr}'
      }
    );
},
```

创建PixDiff对象的时候，可以通过参数对其进行配置。其中basePath指定基准图片的保存位置，diffPath对应的文件夹用于指定图片比对后的差异结果。



建议在创建PixDiff对象的时候，设置baseline为true，这样当第一次运行测试用例还没有基准文件时，测试用例会自动保存截屏作为基准文件。保存的图片名可以通过formatImageName进行任意定制。

（3）在测试用例中，通过调用pix-diff方法checkScreen或checkRegion可以分别对浏览器或者某个元素区域进行匹配。

以下为可能返回的比对结果，其中RESULT_SIMILAR和RESULT_IDENTICAL表示图像相似或一致。由于图像是通过截屏获取到的，考虑到图像的像素质量和比对效果，建议同时使用RESULT_SIMILAR和RESULT_IDENTICAL表示匹配成功。

- RESULT_UNKNOWN

① koola. pix-diff[OL]. [2016]. https://github.com/koola/pix-diff.
② yahoo. Blink-diff[OL]. [2016]. https://github.com/yahoo/blink-diff.

Protractor框架下可以通过插件pix-diff①实现。pix-diff依赖于另一个模块blink-diff②构建，进行图像比较，具体步骤如下。

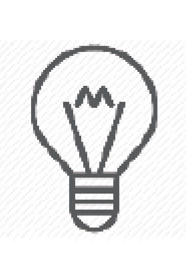
（1）在命令控制台执行以下命令安装插件。

```
npm install -save-dev pix-diff
```

（2）修改protractor配置文件，添加以下代码启用并设置插件。

```
onPrepare: function() {
   var PixDiff = require('pix-diff');
   browser.pixDiff = new PixDiff({
      basePath: './screenshots/',
      baseline: true,
      diffPath: './screenshots/',
      formatImageName: '{tag}-{browserName}-{width}x{height}-dpr-{dpr}'
     }
   );
},
```

创建PixDiff对象的时候，可以通过参数对其进行配置。其中basePath指定基准图片的保存位置，diffPath对应的文件夹用于指定图片比对后的差异结果。



建议在创建PixDiff对象的时候，设置baseline为true，这样当第一次运行测试用例还没有基准文件时，测试用例会自动保存截屏作为基准文件。保存的图片名可以通过formatImageName进行任意定制。

（3）在测试用例中，通过调用pix-diff方法checkScreen或checkRegion可以分别对浏览器或者某个元素区域进行匹配。

以下为可能返回的比对结果，其中RESULT_SIMILAR和RESULT_IDENTICAL表示图像相似或一致。由于图像是通过截屏获取到的，考虑到图像的像素质量和比对效果，建议同时使用RESULT_SIMILAR和RESULT_IDENTICAL表示匹配成功。

- RESULT_UNKNOWN

① koola. pix-diff[OL]. [2016]. https://github.com/koola/pix-diff.

② yahoo. Blink-diff[OL]. [2016]. https://github.com/yahoo/blink-diff.

- RESULT_DIFFERENT
- RESULT_SIMILAR
- RESULT_IDENTICAL

以下为测试用例的示例代码。

```
var listpage = require('../page-objects/phone-list-page.js');
var BlinkDiff = require('blink-diff');
describe('View: Phone list', function() {
   beforeEach(function() {
     listpage.goHome();
   });
   it('demo checkScreen', function() {
     listpage.setQueryField('4G')
       .then(() => browser.pixDiff.checkScreen('demo-checkScreen'))
       .then(result => expect( (result.code === BlinkDiff.RESULT_IDENTICAL) ||
             (result.code === BlinkDiff.RESULT_SIMILAR) ).toBeTruthy() );
   });
});
```

（4）在比对的时候可以通过thresholdType指定匹配方式，通过threshold指定匹配阈值。例如BlinkDiff.THRESHOLD_PIXEL表示基于像素进行匹配，BlinkDiff.THRESHOLD_PERCENT表示基于百分比进行匹配，BlinkDiff默认基于像素进行比较。

```
it('demo threshold', function() {
     listpage.orderByName()
       .then(() => browser.pixDiff.checkScreen('demo-threshold',
             {thresholdType: BlinkDiff.THRESHOLD_PERCENT, threshold: 0.2}))
       .then(result => expect( (result.code === BlinkDiff.RESULT_IDENTICAL) ||
             (result.code === BlinkDiff.RESULT_SIMILAR) ).toBeTruthy() );
   });
```

BlinkDiff支持丰富的参数设置，包括是否启用调试模式等，可以参考https://github.com/yahoo/blink-diff获取更多选项的详细说明。

13.6 任务自动化

与单元测试一样，在最佳实践中同样建议把自动化测试与任务工具集成起来，这样可以通过配置文件设置任务的依赖关系与实施细节，从而简化任务流程。

13.6.1 与gulp集成

本节基于插件gulp-protractor①介绍如何通过任务构建工具gulp驱动Protractor自动化测试，关于gulp的详细使用说明请参考第5章的相关内容。

（1）在命令控制台执行以下命令安装插件。

```
npm install -save-dev gulp-protractor
```

（2）修改Protractor配置文件，基于测试用例的种类与用途，通过suites字段对测试用例进行分类。

protractor.local.conf.js示例代码如下：

```
exports.config = {
  directConnect: true,
  suites: {
    smoke: 'specs/phone-list.js',
    full: 'specs/*.js'
  },
  capabilities: {
    'browserName': 'chrome'
  },
  baseUrl: 'http://angular.github.io/',
  framework: 'jasmine2'
};
```

① Müller & Sohn Digitalmanufaktur GmbH. gulp-protractor[OL]. [2016]. https://github.com/mllrsohn/gulp-protractor.

以上配置指定了两个测试suites，分别是冒烟测试和完整测试，它们对应于不同的测试用例范围。

（3）添加Gulpfile.js文件，设置两个任务e2e-smoke和e2e-full，分别用于执行冒烟测试和完整测试。

```
var gulp = require('gulp');
var gp = require('gulp-protractor');
// Setting up the smoke test task
gulp.task('e2e-smoke', [], function(cb) {
   gulp.src([]).pipe(gp.protractor({
      configFile: 'protractor.local.conf.js',
      args: ['--suite', 'smoke']
   })).on('error', function(e) {
      console.log(e)
   }).on('end', cb);
});
// Setting up the full test task
gulp.task('e2e-full', [], function(cb) {
   gulp.src([]).pipe(gp.protractor({
      configFile: 'protractor.local.conf.js',
      args: ['--suite', 'full']
   })).on('error', function(e) {
      console.log(e)
   }).on('end', cb);
});
```

在以上示例代码中，任务构建通过管道设置Protractor启动选项，其中configFile用于设置配置文件。args用于设置所需参数，支持所有protractor的命令行参数。

（4）在命令控制台执行以下命令，分别进行冒烟测试和完整测试（确保已经下载浏览器驱动）。

```
gulp e2e-smoke
gulp e2e-full
```

（5）自动化测试依赖于浏览器驱动和Selenium Server，通过任务构建可以在启动测试用例前自动更新浏览器驱动，从而避免手工干预。示例代码如下：

```
var gulp = require('gulp');
var gp = require('gulp-protractor');
// Downloads the selenium webdriver
gulp.task('webdriver_update', gp.webdriver_update);
// Setting up the smoke test task
gulp.task('e2e-smoke', ['webdriver_update'], function(cb) {
   gulp.src([]).pipe(gp.protractor({
      configFile: 'protractor.local.conf.js',
      args: ['--suite', 'smoke']
   })).on('error', function(e) {
      console.log(e)
   }).on('end', cb);
});
// Setting up the full test task
gulp.task('e2e-full', ['webdriver_update'], function(cb) {
   gulp.src([]).pipe(gp.protractor({
      configFile: 'protractor.local.conf.js',
      args: ['--suite', 'full']
   })).on('error', function(e) {
      console.log(e)
   }).on('end', cb);
});
// Setting up the default task
gulp.task('default', ['e2e-full']);
```

以上示例代码添加了一个新任务webdriver_update，并将其添加到另外两个任务的依赖中。这样，当开始执行测试用例的时候，gulp会发现webdriver_update是一个被依赖的任务，需要先执行。

以上代码的最后一行将完整测试设置为默认任务，这样在命令控制台执行命令gulp即可直接运行完整测试。

13.6.2　npm脚本

除了读者已经熟知的gulp和Grunt，npm允许在package.json文件内使用scripts字段定义脚本命令[①]，执行命令npm run会新建一个Shell并在这个Shell里执行指定的脚本命令，从而也可以作为构建工具使用。注意，npm run命令新建的这个Shell，会在当前目录的node_modules/.bin子目录中加入PATH环境变量，执行结束后，再将PATH变量恢复。这也意味着，当前目录内的node_modules/.bin子目录里的所有脚本，都可以直接用脚本名调用，而不必额外添加路径[②]。与gulp和Grunt比较而言，npm可以不依赖于任何额外的构建插件，对调用CLI或者Shell脚本提供了原生支持。当然，在实际使用中，读者既可以使用npm直接构建任务，也可以通过调用gulp或Grunt完成相同的功能。

表13-1是npm已经预定义好的常用任务，完整的任务列表请参考网址https://docs.npmjs.com/misc/scripts中所述。以任务test为例，用户仍然可以通过执行npm run test命令来运行，但作为预定义任务，也可以直接通过调用npm test来运行。

表13-1　npm预定义任务

任务名称	说　明
preinstall, install, postinstall	调用npm install后按序执行，定义安装相关的任务
pretest, test, posttest	调用npm test后按序执行，定义测试相关的任务
prestart, start, poststart	调用npm start后按序执行，定义启动相关的任务

以下package.json示例代码定义了两个任务test和e2e-smoke，均通过调用gulp启动自动化测试。其中test是预定义任务，e2e-smoke是自定义任务。

```
{
  "name": "phonecat-e2e",
```

① npm. npm-scripts[OL]. [2016]. https://docs.npmjs.com/misc/scripts.

② Keith Cirkel. How to use npm as a Build Tool[OL]. 2014. https://www.keithcirkel.co.uk/how-to-use-npm-as-a-build-tool/.

```
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "gulp",
    "e2e-smoke": "gulp e2e-smoke"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "gulp": "^3.9.1",
    "gulp-protractor": "^3.0.0",
    "protractor": "^4.0.11"
  }
}
```

在命令控制台执行以下命令即可启动任务，如图13-8所示。

```
npm test
npm run e2e-smoke
```

```
Administrator: Command Prompt

C:\phonecat-e2e>npm test

> phonecat-e2e@1.0.0 test C:\phonecat-e2e
> gulp

[11:06:25] Using gulpfile C:\phonecat-e2e\gulpfile.js
[11:06:25] Starting 'webdriver_update'...
[11:06:26] I/update - chromedriver: file exists C:\phonecat-e2e\node_modules\pro
tractor\node_modules\webdriver-manager\selenium\chromedriver_2.25win32.zip
[11:06:26] I/update - chromedriver: unzipping chromedriver_2.25win32.zip
[11:06:26] I/update - chromedriver: v2.25 up to date
[11:06:26] I/update - selenium standalone: file exists C:\phonecat-e2e\node_modu
les\protractor\node_modules\webdriver-manager\selenium\selenium-server-standalon
e-2.53.1.jar
[11:06:26] I/update - selenium standalone: v2.53.1 up to date
[11:06:28] I/update - geckodriver: file exists C:\phonecat-e2e\node_modules\prot
ractor\node_modules\webdriver-manager\selenium\geckodriver-v0.11.1-win64.zip
[11:06:28] I/update - geckodriver: unzipping geckodriver-v0.11.1-win64.zip
[11:06:30] I/update - geckodriver: vv0.11.1 up to date
[11:06:36] Finished 'webdriver_update' after 11 s
[11:06:36] Starting 'e2e-full'...
[11:06:36] I/direct - Using ChromeDriver directly...
[11:06:36] I/launcher - Running 1 instances of WebDriver
Started
.......


7 specs, 0 failures
Finished in 22.824 seconds
[11:07:02] I/launcher - 0 instance(s) of WebDriver still running
[11:07:02] I/launcher - chrome #01 passed
[11:07:02] Finished 'e2e-full' after 26 s
[11:07:02] Starting 'default'...
[11:07:02] Finished 'default' after 33 μs
```

图13-8　npm驱动测试

第14章 分布式自动化测试

尽管Protractor可以通过Selenium Server远程执行测试脚本，但一个Selenium Server无法同时支持多种异构的测试环境，例如Windows上的IE和Linux上的Chrome，也无法兼顾浏览器的多种版本。另外，随着测试用例的不断累积，每次测试所需的时间也会越来越长，当用例数量达到一定数量级后会直接影响到测试性能。所以，在大型应用的开发环境内，需要一种能够兼顾多种异构环境、多个测试用例可以并发执行的分布式测试平台。

本章将介绍：

- 分布式测试概述
- 基于Selenium Grid的分布式测试
- 基于云计算的分布式测试
- 配置共享

14.1 分布式测试概述

分布式测试在局域网内或通过互联网，把分布于不同地点、能够独立完成测试功能的计算机资源连接起来，从而达到共享计算资源、分散操作、集中管理和统一调度的效果。与单机模式比较，分布式测试有以下优点：

1. 互通性

多个测试节点之间的互连互通实现了局域网或互联网内的资源共享，是分布式测试系统的底层支撑结构。

2. 操作系统无关性

成熟的分布式测试平台需要兼容多种异构环境，无论是Windows、Linux还是Mac OS

都可以无缝集成。

3. 可伸缩性

当计算资源不够或需要新的异构测试环境时，可以方便地将新的计算资源加入到已有的测试平台内，而无需重新搭建完整的测试环境。

4. 并发性

分布式测试系统本质上是一个实时系统，基于统一调度的任务分发机制保证测试任务能够根据环境要求，实时分发到合适的计算节点并发执行。

5. 容错性

分布式测试的计算节点互不影响，任何出问题的节点不会影响其他节点的操作及整个测试平台的运行。

对分布式测试平台而言，核心是流程控制，即系统需要实时调度计算资源并能够方便地监视和操纵测试过程。因此，分布式测试系统一般采用集中管理的分布式策略，即由一台中心计算机控制若干台受控计算机的执行。整个测试过程和资源管理由中心计算机来完成，它掌握整个测试环境的状态，发出调度命令。

14.2 基于Selenium Grid的分布式测试

Selenium Grid是一个基于Java构建的分布式测试管理系统，兼容当前所有的主流操作系统。

Selenium Grid架构中包含两个角色，分别是中央节点（Hub）和工作节点（Node）。如图14-1所示，每个工作节点可以运行在不同的操作系统中并支持不同的浏览器，是实施测试的实际资源；中央节点用于管理各个工作节点的注册和状态信息，接受远程WebDriver测试脚本请求，根据测试脚本的环境要求调度合适的节点资源，并把随后的测试命令转发到对应的工作节点执行。

由于工作节点可以配置不同的运行环境，针对某种使用频率较高的测试环境可以按需分配更多的计算资源，Selenium Grid大大提高了测试用例的执行效率并节省了硬件资源的消耗。

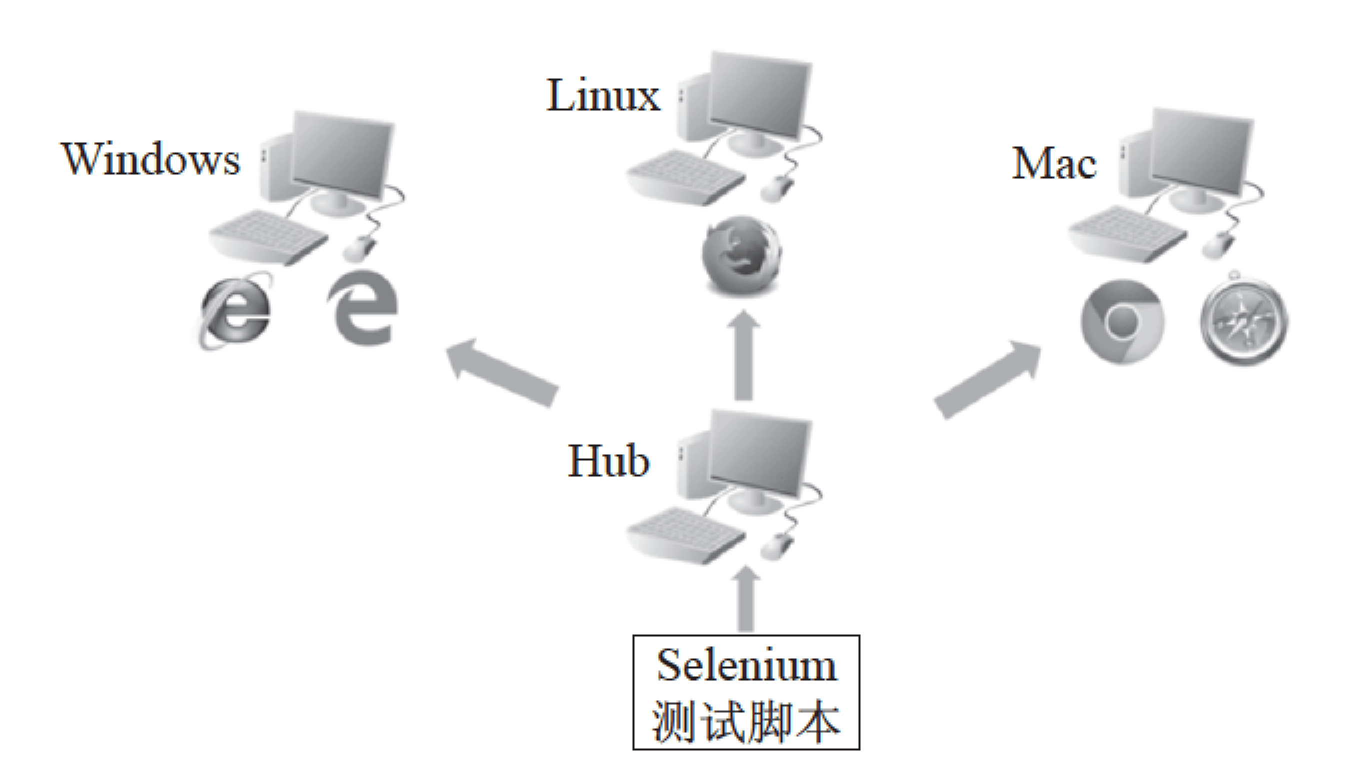


图14-1　Selenium Grid原理

14.2.1　启动中央节点

在命令控制台执行以下命令启动Selenium Grid的中央节点，其默认运行于4444端口。参数-role hub表明当前启动的是一个中央节点。

```
java -jar selenium-server-standalone-2.53.1.jar -role hub
```

命令运行结果如图14-2所示，后续工作节点将通过网址http://192.168.2.51:4444/grid/register/进行注册。

```
Administrator: Command Prompt - java  -jar selenium-server-standalone-2.53.1.jar -role hub

C:\selenium>java -jar selenium-server-standalone-2.53.1.jar -role hub
10:04:25.895 INFO - Launching Selenium Grid hub
2016-12-08 10:04:26.716:INFO::main: Logging initialized @978ms
10:04:26.732 INFO - Will listen on 4444
10:04:26.763 INFO - Will listen on 4444
2016-12-08 10:04:26.763:INFO:osjs.Server:main: jetty-9.2.z-SNAPSHOT
2016-12-08 10:04:26.794:INFO:osjsh.ContextHandler:main: Started o.s.j.s.ServletContextHandler@6fb0d3ed{/,null,A
2016-12-08 10:04:26.810:INFO:osjs.ServerConnector:main: Started ServerConnector@382db087{HTTP/1.1}{0.0.0.0:4444
2016-12-08 10:04:26.810:INFO:osjs.Server:main: Started @1074ms
10:04:26.810 INFO - Nodes should register to http://192.168.2.51:4444/grid/register/
10:04:26.810 INFO - Selenium Grid hub is up and running
```

图14-2　启动中央节点

14.2.2　注册工作节点

中央节点启动后，用户需要将工作节点注册到该中央节点。

与中央节点类似，工作节点仍然通过selenium-server-standalone二进制文件进行注册，参数-role node表明当前注册的是一个工作节点，而且需要通过hub参数提供中央节点的注

册地址。参数browser用于指定该工作节点支持的浏览器信息，包括浏览器的名称、版本和操作系统等。

如图14-3所示，在命令控制台执行以下命令将注册一个支持Chrome的工作节点，该工作节点监听默认端口5555，接受中央节点的测试指令。

```
java -jar selenium-server-standalone-2.53.1.jar -role node -hub http://192.168.2.51:4444/
grid/register -browser "browserName=chrome,version=ANY" -Dwebdriver.chrome.driver=
chromedriver_2.25.exe
```

同一台机器可以支持多个工作节点，但需要运行于不同的端口上用于监听中央节点的指令。例如以下命令在5556端口创建了另一个工作节点，可支持47.0.2版的Firefox。

```
java -jar selenium-server-standalone-2.53.1.jar -role node -hub http://192.168.2.51:4444/
grid/register -browser "browserName=firefox,version=47.0.2" -port 5556
```

```
Administrator: Command Prompt - java -jar selenium-server-standalone-2.53.1.jar -role node -hub http://192.168.2.51:4444/grid/register -browser "bro...

C:\selenium>java -jar selenium-server-standalone-2.53.1.jar -role node -hub http://192.168.2.51:4444/grid/register -brow
ser "browserName=chrome,version=ANY" -Dwebdriver.chrome.driver=chromedriver_2.25.exe
10:16:09.458 INFO - Launching a Selenium Grid node
10:16:09.520 INFO - Adding browserName=chrome,version=ANY
Setting system property webdriver.chrome.driver to chromedriver_2.25.exe
10:16:10.928 INFO - Java: Oracle Corporation 25.112-b15
10:16:10.928 INFO - OS: Windows 10 10.0 amd64
10:16:10.928 INFO - v2.53.1, with Core v2.53.1. Built from revision a36b8b1
10:16:10.975 INFO - Driver class not found: com.opera.core.systems.OperaDriver
10:16:10.975 INFO - Driver provider com.opera.core.systems.OperaDriver is not registered
10:16:10.991 INFO - Driver provider org.openqa.selenium.safari.SafariDriver registration is skipped:
registration capabilities Capabilities [{browserName=safari, version=, platform=MAC}] does not match the current platfor
m WIN10
10:16:10.991 INFO - Driver class not found: org.openqa.selenium.htmlunit.HtmlUnitDriver
10:16:10.991 INFO - Driver provider org.openqa.selenium.htmlunit.HtmlUnitDriver is not registered
10:16:11.006 INFO - Version Jetty/5.1.x
10:16:11.006 INFO - Started HttpContext[/selenium-server,/selenium-server]
10:16:11.006 INFO - Started org.openqa.jetty.jetty.servlet.ServletHandler@971d0d8
10:16:11.022 INFO - Started HttpContext[/wd,/wd]
10:16:11.022 INFO - Started HttpContext[/selenium-server/driver,/selenium-server/driver]
10:16:11.022 INFO - Started HttpContext[/,/]
10:16:11.022 INFO - Started SocketListener on 0.0.0.0:5555
10:16:11.022 INFO - Started org.openqa.jetty.jetty.Server@670b40af
10:16:11.022 INFO - Selenium Grid node is up and ready to register to the hub
10:16:11.053 INFO - Starting auto registration thread. Will try to register every 5000 ms.
10:16:11.053 INFO - Registering the node to the hub: http://192.168.2.51:4444/grid/register
10:16:11.287 INFO - The node is registered to the hub and ready to use
```

图14-3 注册工作节点

一个工作节点可以支持多种浏览器，需要用多个browser参数来指定。例如以下命令创建的工作节点同时支持Edge和IE。

```
java -jar selenium-server-standalone-2.53.1.jar -role node -hub http://192.168.2.51:4444/
grid/register -browser "browserName=MicrosoftEdge,version=ANY" -browser "browserName=internet
explorer,version=ANY" -Dwebdriver.ie.driver=IEDriverServer.exe -Dwebdriver.edge.driver=
MicrosoftWebDriver.exe
```

所有已经注册的工作节点会显示在中央节点的管理页面（本例为http://192.168.2.51:4444/grid/console）。中央节点能够实时监控各个工作节点的运行情况，任何离线的工作节点不会影响到其他节点的运行与调度。如图14-4所示，可以看到中央节点注册在http://192.168.2.51:4444上，它注册了3个工作节点：注册在http://192.168.2.55:5555上的工作节点，支持Chrome测试；注册在http://192.168.2.55:5556上的工作节点，支持Firefox测试；注册在http://192.168.2.54:5555上的工作节点，支持IE和Edge测试。

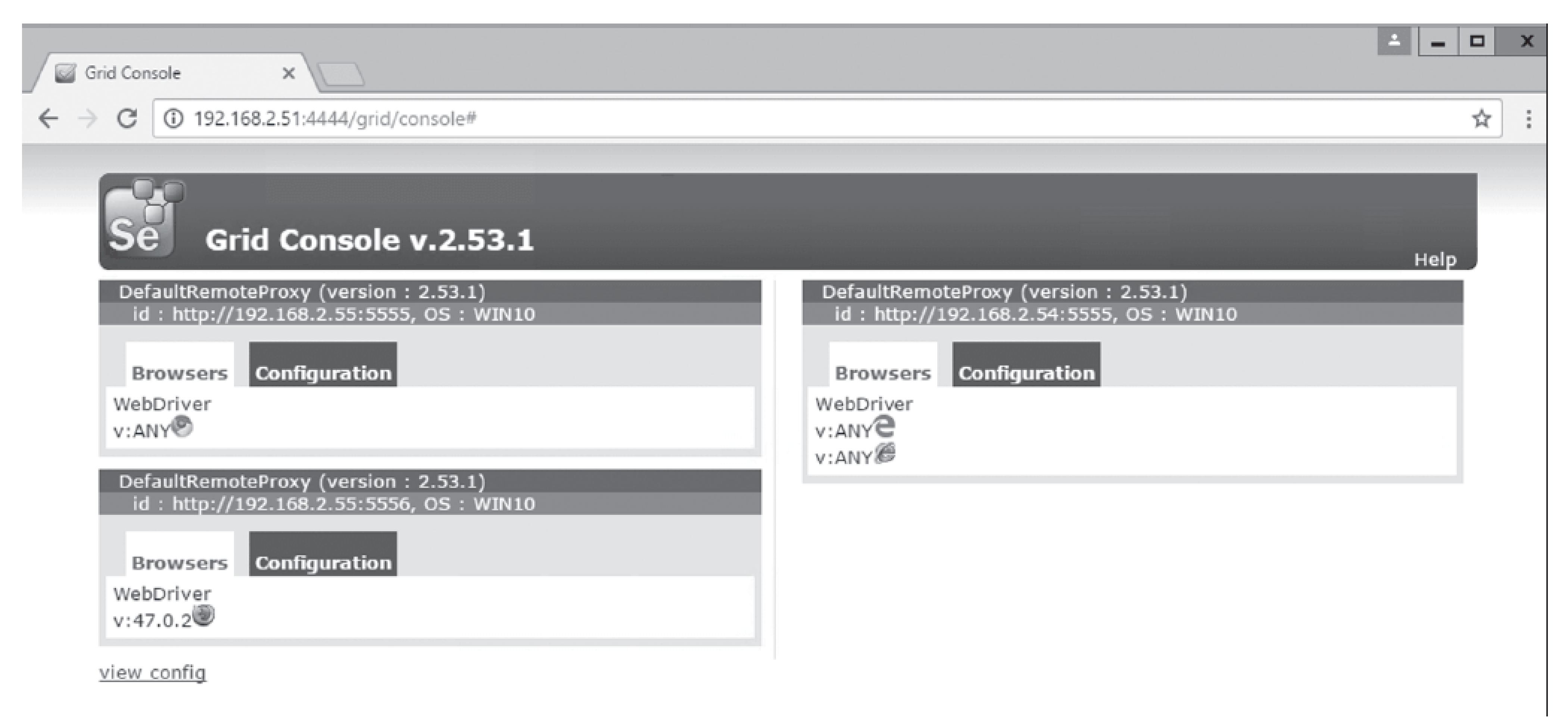


图14-4　Selenium Grid管理页面

14.2.3　执行测试

如以下代码所示，只需在Protractor配置文件中设置seleniumAddress字段为中央节点的地址，就可以通过Selenium Grid远程执行测试脚本。对于Protractor测试用例而言，并不需要关心运行测试代码的是一台Selenium Server还是Selenium Grid中的某台工作节点。

```
exports.config = {
  directConnect: false,
  seleniumAddress: 'http://192.168.2.51:4444/wd/hub',
  specs: [
    'specs/*.js'
  ],
  multiCapabilities: [
    {browserName: 'firefox', version:'47.0.2'},
```

```
    {browserName: 'MicrosoftEdge',elementScrollBehavior: 1, nativeEvents: false},

    {browserName: 'internet explorer'},

    {browserName: 'chrome'}],

  baseUrl: 'http://angular.github.io/',

  framework: 'jasmine2',

};
```

中央节点根据配置文件中指定的浏览器类型和版本，遍历工作节点并检查是否有符合要求的可用节点，然后由选定的工作节点执行测试用例。

测试结束后，测试结果会显示到终端，如图14-5所示。测试用例并不需要了解究竟是哪个工作节点执行了测试任务。

```
Administrator: Command Prompt

C:\phonecat-e2e>protractor protractor.grid.conf.js
[10:41:15] I/launcher - Running 4 instances of WebDriver
....................[10:42:09] I/testLogger -
------------------------------------

[10:42:09] I/testLogger - [MicrosoftEdge #11] PID: 2040
[MicrosoftEdge #11] [10:41:16] I/hosted - Using the selenium server at http://192.168.2.51:4444/wd/hub
[MicrosoftEdge #11] Started
[MicrosoftEdge #11] .......
[MicrosoftEdge #11]
[MicrosoftEdge #11]
[MicrosoftEdge #11] 7 specs, 0 failures
[MicrosoftEdge #11] Finished in 38.843 seconds

[10:42:09] I/testLogger -

[10:42:09] I/launcher - 3 instance(s) of WebDriver still running
[10:42:10] I/testLogger -
------------------------------------

[10:42:10] I/testLogger - [internet explorer #21] PID: 2240
[internet explorer #21] [10:41:16] I/hosted - Using the selenium server at http://192.168.2.51:4444/wd/hub
[internet explorer #21] Started
[internet explorer #21] .......
[internet explorer #21]
[internet explorer #21]
[internet explorer #21] 7 specs, 0 failures
[internet explorer #21] Finished in 47.853 seconds

[10:42:10] I/testLogger -

[10:42:10] I/launcher - 2 instance(s) of WebDriver still running
.......[10:42:29] I/testLogger -
------------------------------------

[10:42:29] I/testLogger - [firefox 47.0.2 #01] PID: 2448
[firefox 47.0.2 #01] [10:41:16] I/hosted - Using the selenium server at http://192.168.2.51:4444/wd/hub
[firefox 47.0.2 #01] Started
[firefox 47.0.2 #01] .......
[firefox 47.0.2 #01]
[firefox 47.0.2 #01]
[firefox 47.0.2 #01] 7 specs, 0 failures
[firefox 47.0.2 #01] Finished in 41.556 seconds

[10:42:29] I/testLogger -

[10:42:29] I/launcher - 1 instance(s) of WebDriver still running
[10:42:32] I/testLogger -
------------------------------------

[10:42:32] I/testLogger - [chrome #31] PID: 836
[chrome #31] [10:41:16] I/hosted - Using the selenium server at http://192.168.2.51:4444/wd/hub
[chrome #31] Started
[chrome #31] .......
[chrome #31]
[chrome #31]
[chrome #31] 7 specs, 0 failures
[chrome #31] Finished in 44.512 seconds

[10:42:32] I/testLogger -

[10:42:32] I/launcher - 0 instance(s) of WebDriver still running
[10:42:32] I/launcher - MicrosoftEdge #11 passed
[10:42:32] I/launcher - internet explorer #21 passed
[10:42:32] I/launcher - firefox47.0.2 #01 passed
[10:42:32] I/launcher - chrome #31 passed
```

图14-5　基于Selenium Grid执行测试

14.3　基于云计算的分布式测试

近几年，经过不断的积累和持续探索，云计算达到了前所未有的热度，已经开始成为全球信息产业发展的主题，这是不断提高的计算能力以及快速普及的数字化产业革命催生的新的商业模式。

基于美国国家标准与技术研究院（NIST）的定义，云计算是一种按使用量付费的模式，这种模式提供可用的、便捷的、按需的网络访问。当使用者需要计算资源的时候，只需要进入计算资源共享池（资源包括网络、服务器、存储、应用软件、服务）加以简单配置，期望的资源就能够被快速提供。整个过程中，使用者只需投入很少的管理工作，或与服务供应商进行很少的交互。

在传统的本地模式下，公司需要充分考虑底层网络、数据存储、负载均衡和管理运维等事务，公司虽然对所有资源有最充分的管理权限，但IT资源的采购和维护费用高昂。特别是建立初期，容量往往高于实际的业务负载，造成了服务器闲置以及资源的浪费。同时，随着公司业务的增长，后期硬件资源又可能会出现不够用的情况，使之成为制约公司发展的瓶颈。无论是资源过度配置还是供给不足，都是IT能力供需之间的矛盾，而这正是云计算致力解决的问题。

与传统模式不同，云计算强调的是资源共享。用户仅需订阅云服务商的计算资源即可完成本地模式下的相同业务，因为基础设施是由云服务商维护和管理。云服务商可以根据用户的需求实时部署资源，用户也按需付费，从而实现定制化成本以及IT设施的利用率最大化。对用户而言，将本地模式转换成云计算模式，可以大幅节省人力和物力，帮助企业快速增长。

在云计算的环境下，软件开发工具、环境、工作模式也正在发生巨大转变，这也要求软件测试的工具、环境、工作模式也随之发生相应的转变。以基于Selenium Grid的自动化测试为例，对一个企业而言，提供所有的测试环境包括操作系统和浏览器是不现实的，而使用云上的测试环境，则可以大大拓展本地测试环境的局限性，通过云实现协同管理、知识共享以及测试复用。

目前市场上提供应用程序测试平台的云服务商很多，例如Sauce Labs和BrowserStack。其中，Sauce Labs的联合创始人正是大名鼎鼎的Selenium发明人Jason Huggins。

Sauce Labs的云测试使用Selenium WebDriver作为底层测试解决方案，可以对网络浏览器进行自动化验收测试。基于Sauce Labs，开发人员可以测试所有主流操作系统上的所有

主流浏览器，包括IE、Firefox、Safari、Chrome等。在测试时可以对错误进行截屏和视频记录。下面演示使用Protractor驱动Sauce Labs进行自动化测试的方法。

（1）在Sauce Labs上注册后，可以登录到其管理界面。如图14-6所示，在用户设置页面，可以获得秘钥（Access Key），该秘钥与用户名配对用于唯一地标识使用者的身份。

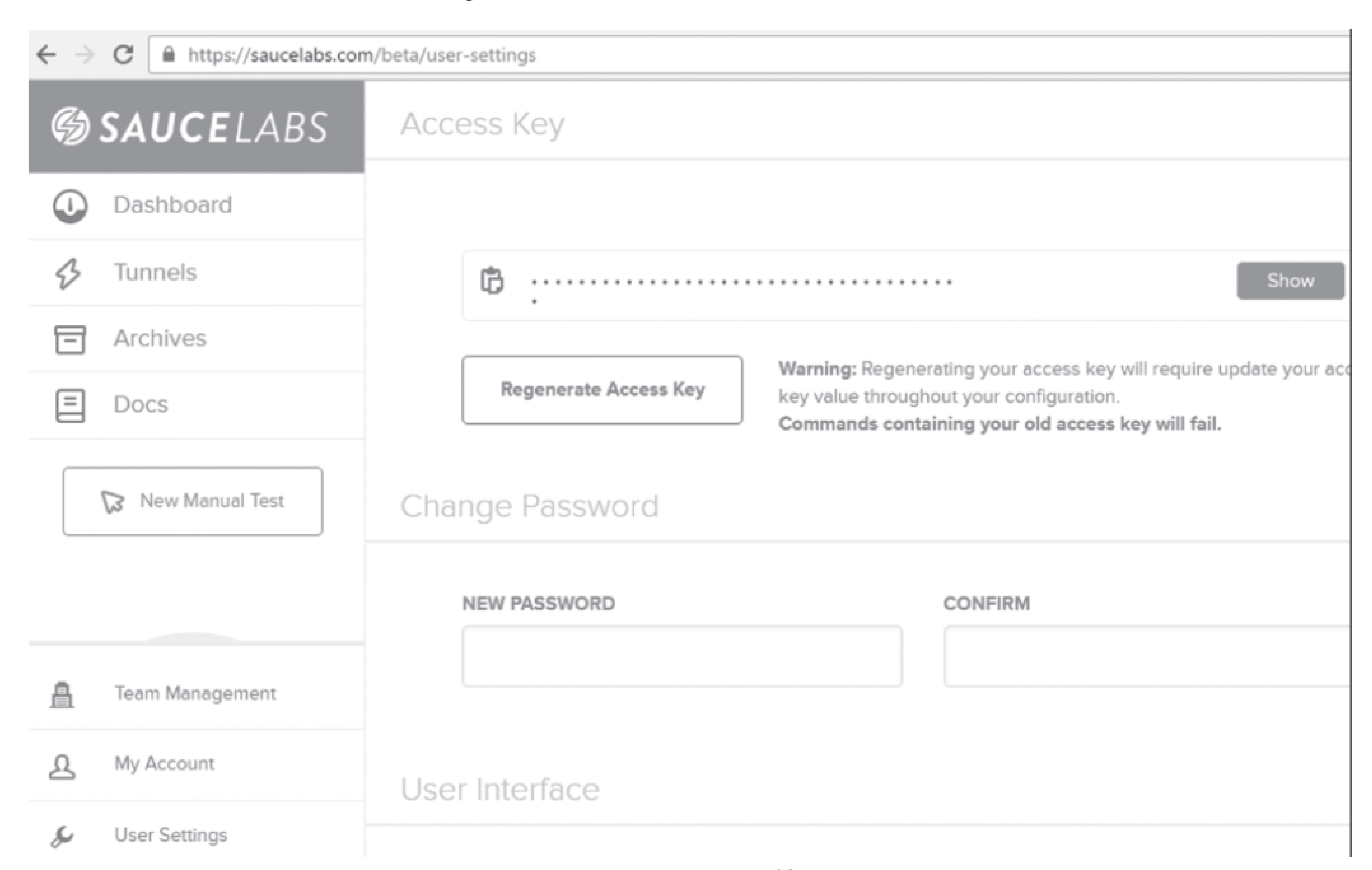


图14-6　Sauce Labs管理界面

（2）Protractor配置文件提供了关键字sauceUser和sauceKey，将用户名与秘钥填入即可在Sauce Labs上远程执行测试用例。秘钥的作用是用户身份验证，需要妥善保管。



为了避免秘钥被暴露到版本控制库，建议基于Sauce Labs的测试只运行于集成环境，而不要运行于开发环境。为此，可以将用户名和秘钥保存到集成服务器的环境变量内以避免泄露。

```
exports.config = {
  sauceUser:process.env.Sauce_User,
  sauceKey:process.env.Sauce_Key,
  specs: [
    'specs/*.js'
  ],
  multiCapabilities: [
```

```
    {browserName: 'firefox', version:'47'},
    {browserName: 'MicrosoftEdge',elementScrollBehavior: 1, nativeEvents: false},
    {browserName: 'internet explorer',platform:'windows 10'},
    {browserName: 'chrome',platform:'windows'},
    {browserName: 'chrome',platform:'linux'},
    {browserName: 'safari',platform:'mac'},
    {browserName: 'safari',platform:'windows 7'}],
  baseUrl: 'http://angular.github.io/',
  framework: 'jasmine2',
};
```

（3）Sauce Labs会基于Protractor配置文件内对操作系统和浏览器类型的声明选择计算资源，并行地运行测试用例。所有的测试历史记录可以通过管理界面查询获得，如图14-7所示，包括测试运行的操作系统、浏览器类型和版本，以及测试结果。

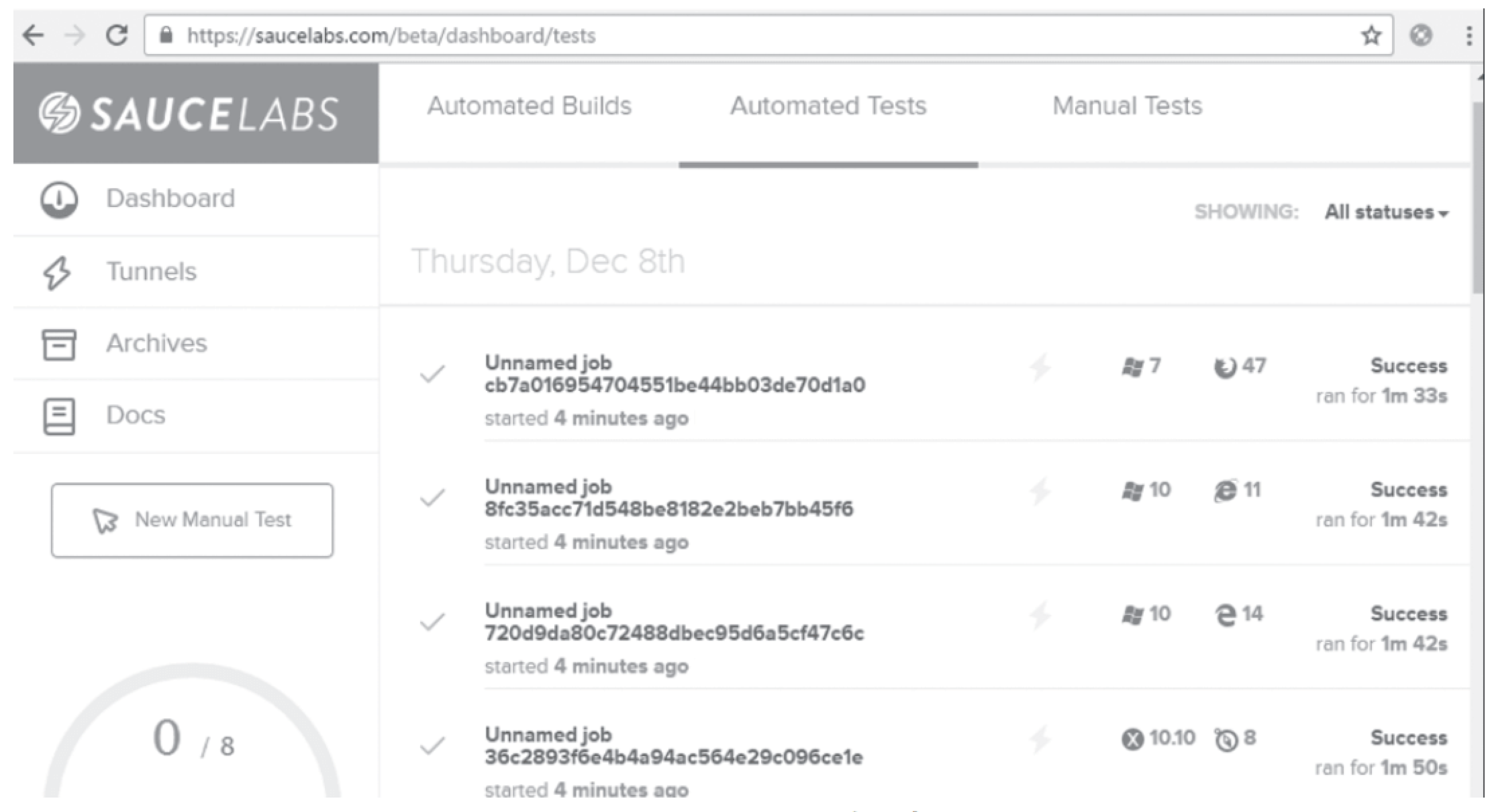


图14-7 Sauce Labs测试历史记录

（4）针对每一个测试用例，Sauce Labs都提供了视频记录、命令记录、服务端记录和测试配置，如图14-8所示，特别是命令记录和服务端记录还可以方便地帮助用户进行排错工作。

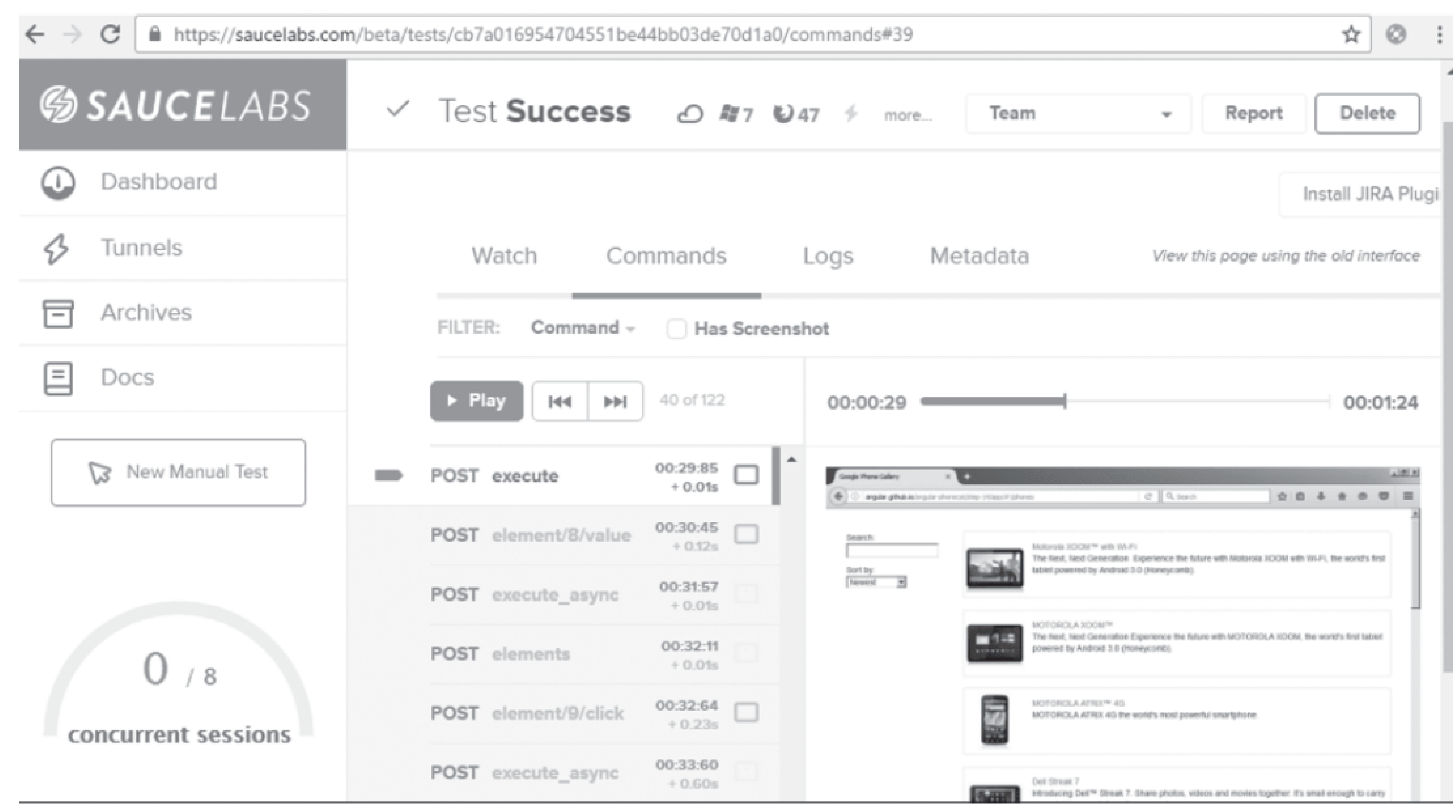


图14-8　Sauce Labs测试报告

（5）被测网站在正式发布之前无法通过互联网访问，或者被测网站位于企业防火墙之后，对于这类情况，可以使用Sauce Connect代理服务器通过为Sauce Labs的虚拟机与被测网站之间建立连接通道，达到测试的目的。

14.4　配置共享

Protractor的自动化测试非常灵活，既可以在本机上直接运行测试用例，也可以通过远程的Selenium Server或Selenium Grid进行测试，甚至可以使用云服务商提供的托管环境进行测试。用户只需要修改配置文件即可做到，非常方便。

以上3种方式各有优点也各有其适用的情况。一般来说：

- 本机直连是速度最快效率最高的，适合开发人员在开发过程中对脚本行为进行验证与修改。
- 在公司内搭建专门的测试服务器，适合针对产品的主要运行环境进行集成测试。
- 对于大型或使用基数大的应用，公司的测试服务器往往难以覆盖所有的测试环境，这时候可以考虑基于云计算的分布式测试服务。

这3种测试方案的区别主要在于Protractor的部分配置不同。从最佳实践的角度来说，

如果能把测试用例的声明、测试报告的设置等一致的部分在3个配置文件中进行直接共享，则可以减少配置的维护成本，防止出现误改或漏改的情况。

以下示例代码中，创建protractor.shared.conf.js文件作为配置的共享部分，被其他3个配置文件引用。

以下为protractor.shared.conf.js文件的示例代码：

```
exports.config = {
  specs: ['specs/*.js'],
  baseUrl: 'http://angular.github.io/',
  framework: 'jasmine2',
  onPrepare: function() {
    var jasmineReporters = require('jasmine-reporters');
    jasmine.getEnv().addReporter(new jasmineReporters.JUnitXmlReporter({
        consolidateAll: true,
        savePath: 'testresults',
        filePrefix: 'xmloutput'
    }));
  }
};
```

以下为protractor.local.conf.js文件的示例代码：

```
var config = require('./protractor.shared.conf.js').config;
config.directConnect = true;
config.capabilities = {browserName: 'chrome'};
exports.config = config;
```

以下为protractor.remote.conf.js文件的示例代码：

```
var config = require('./protractor.shared.conf.js').config;
config.directConnect = false;
config.seleniumAddress = 'http://192.168.2.51:4444/wd/hub';
```

```
config.multiCapabilities = [
   {browserName: 'firefox', version:'47'},
   {browserName: 'MicrosoftEdge',elementScrollBehavior: 1, nativeEvents: false},
   {browserName: 'internet explorer'},
   {browserName: 'chrome'}];
exports.config = config;
```

以下为protractor.cloud.conf.js文件的示例代码：

```
var config = require('./protractor.shared.conf.js').config;
config.sauceUser = process.env.Sauce_User;
config.sauceKey = process.env.Sauce_Key;
config.multiCapabilities = [
   {browserName: 'firefox', version:'47'},
   {browserName: 'MicrosoftEdge',elementScrollBehavior: 1, nativeEvents: false},
   {browserName: 'internet explorer',platform:'windows 10'},
   {browserName: 'chrome',platform:'windows'},
   {browserName: 'chrome',platform:'linux'},
   {browserName: 'safari',platform:'mac'},
   {browserName: 'safari',platform:'windows 7'}];
exports.config = config;
```

集成篇

第15章 持续集成概论

随着云计算、大数据、机器学习等新兴技术的发展，企业在向最终用户交付以及维护安全、高质量软件和服务方面，面临着越来越大的压力。持续集成以敏捷开发为基础，通过整合公司的产品开发和运维部门，让整个软件交付生命周期中的所有参与者，在持续交付的推动下，通过不断的反馈，最终实现产品全生命周期的高效运行。

作为不断发展的开发模式，持续集成不只是简单的技术变革，它在改善产品性能、软件质量以及提升用户体验方面的作用越来越显著。

本章将介绍：

- 开发流程自动化
- 持续集成的功能特征
- 如何实施持续集成
- 选择持续集成工具

15.1 开发流程自动化

事实证明传统的开发模式仅适用于小型，外部依赖较少的项目，随着软件开发复杂度的不断提高，团队开发成员间能否更好的协同工作以确保软件开发的质量，能否通过流程管理解决软件开发的上下游协作已成为开发过程中不可回避的问题。尤其是近些年来，敏捷开发模式在软件工程领域得到广泛应用，软件开发急需一种自我管理，自我适应，让开发自动化起来的新模式。

15.1.1　什么是持续集成

很多人都听说过敏捷方法，这个概念带来了改变团队组织的工作方式、适应不断变化的需求的方式以及发布软件的方式。

持续集成（Continuous Integration，简称CI）正是为敏捷开发而创建的。根据Martin Fowler 的观点，持续集成是一种软件开发实践，要求团队成员经常集成他们的工作，每个人至少每天集成一次，这导致每天有多个集成。集成是通过自动化构建进行的，这些构建运行回归测试，以尽快检测软件缺陷。Martin Fowler建议通过持续集成达到以下目标：

- 任何人在任何地点，任何时间可以构建整个项目。
- 在持续集成构建过程中，每一个测试都必须被执行。
- 在持续集成构建过程中，每一个测试都必须通过。
- 持续集成构建的结果是可以发布的软件包。
- 当以上任何一点不能满足时，整个团队的主要任务是优先解决这个问题。

作为一种软件开发实践，Martin Fowler对如何通过持续集成提高软件开发效率并保障软件开发质量提供了理论基础。经过近十年的不断演化发展，持续集成变成了持续编译、测试、检查和部署源代码的代名词。这意味着每当源代码管理库中的代码发生改变时，都要执行新的构建。开发团队发现，这种以较小增量不断迭代的开发模式能够让集成问题大幅减少，更快的交付有竞争力的软件产品。

15.1.2　持续集成的价值

持续集成倡导团队开发成员经常集成他们的工作，甚至每天都进行多次集成，而每次的集成都是通过自动化构建进行的。这里的构建是编译、部署、测试、审查和反馈的一组流程，自动化的构建意味着整个流程不需要任何用户的手工干预，也叫作无人值守的过程。持续集成的优点包括：

1. 及早发现缺陷

结合测试驱动的开发理论构建测试用例，然后编写功能代码。随着每一次新代码的添加，将其对应的测试用例也添加到集成工作环境的测试套件中。每天多次进行集成并执行测试和审查，可以确保新增代码不会破坏之前的工作。即使出现了回归缺陷，在代码刚提交到代码控制库就被能被发现，开发人员可以迅速获得通知，而不必等到项目后期才发

现。缺陷发现得越早，修复难度越低，对后续功能的影响越小，这也直接降低了软件开发的成本，让团队能够更快、更高效地开发软件。

2. 减少重复劳动

通过构建自动化，让所有工作流自我驱动，包括代码编译、数据库集成、测试、审查、部署和反馈。这大大减少了开发测试人员的重复劳动，让他们有时间做更多需要动脑筋、有更高探索性的工作。

3. 随时发布可部署的软件

很多软件项目都有一个奇怪而又常见的特征，即在开发过程中，程序在相当长的一段时间里是无法运行的。实际上，由大规模团队开发的软件中，绝大部分在开发过程的前半段基本处于不可用状态。因为没有人有兴趣在开发完成之前运行整个应用，可以想象这时会存在很多潜在问题。开发人员更倾向于最后再去解决这些问题。

持续集成的概念带来了自动化构建和可重复构建，自动化测试和可重复测试。这意味着无论是哪种平台，哪种技术，在任何时间点上团队成员提交的代码都应该能够成功集成。换句话说，持续集成让第一时间发现软件缺陷成为可能，如果任何代码变更导致了该问题，开发人员会立刻得到通知进行软件修复，使任意时间发布可部署的软件成为可能。

相反，不采用持续集成实践的项目可能需要等到交付之前才对软件进行集成部署，这可能导致产品发布的延迟或不能修复某些缺陷。如果急于完成任务，则可能引入新的缺陷，最后导致项目失败。

4. 实现分布式团队协作

软件开发一直以来都主张协作是系统成功开发和交付的关键因素。软件生命周期会涉及到开发、测试和运维等不同的团队。大型项目的团队成员一般不可能坐在同一个办公室工作，甚至可能处于不同的时区。有效的分布式团队协作不仅包括从一个团队到另一个团队的有效移交，还包括对需求、特点和安排的全面协调和理解。

持续集成良好的架构可以支持这种协作，原因是通过共享的代码控制系统和持续反馈，技术人员可以更好地了解他们正在构建的各个组件之间的依赖关系。持续集成可以有效地实现分布式团队的协作沟通，确保创建符合需求的产品，并且快速识别和纠正出现的偏差。

5. 反映实时趋势

持续集成让开发人员能够注意到趋势并进行有效的决策。如果没有真实或最新的数据提供支持，项目就会遇到障碍。传统模式下，项目成员以手工方式收集这些信息，不仅增

加了负担，还很耗时。持续集成系统为项目构建状态和品质指标提供了及时的信息，有些持续集成系统可以报告功能完成度和缺陷率。通过这些，项目成员可以看到产品的整体趋势，包括构建成功或失败、总体品质以及其他相关的项目信息。

6. 建立团队信心

团队的信心来自于质量把控，特别是项目经理为了全方位了解开发进展，需要随时问自己，如果我的产品必须在下周发布，哪些部分将会产生最大风险？这是否是一个高品质的发布？如果有需求变更，需要多少时间进行实施？

基于自动化构建的持续集成让团队成员在任何时候都了解产品的状态，特别是结合测试驱动后，可以实时地知道当前已经完成了什么功能，还有什么缺陷需要修复，当前部署的版本对系统有什么要求。面对加快交付周期的市场压力，确切地了解项目进展在需求不明确或是频繁性变更的产品开发中尤为重要。能够快速地回答以上问题也可以帮助项目经理全面掌握要添加哪些资源、在何处调整产品特性，以及何时重新建立交付日期。这一切都是对质量把控的有效帮助，可以让团队对自己的工作更有信心，从而成功地将产品发布到市场并获得竞争优势。

15.2　持续集成的功能特征

持续集成的核心思想是构建自动化，一般来说包括编译、测试、审计、部署和反馈5个特征。也就是说，持续集成的目的正是把这5个功能自动地持续运转起来。功能越复杂，持续集成的价值也越高。

15.2.1　编译

持续的源代码编译是CI系统中最基本的功能，指的是基于开发人员提交的源代码生成可执行代码。近些年，随着更多语言类型例如JavaScript和CSS的不断兴起，编译的概念与传统的C++、C#等出现了某种程度的变化。这些语言种类虽然不会直接生成二进制文件，但都提供了语法检查等类似功能，也可以认为是一种编译过程。另外，越来越多的公司开始尝试用更高级的语言例如TypeScrip和Less来编译生成JavaScript和CSS，所以编译仍然是CI系统中最基本的特征，也是集成开始的地方。

15.2.2 测试

持续测试是CI系统的基石，也是质量管理的保证。如果有人对你说他的CI系统不包括测试的部分，那么这个CI系统只能是无源之水，无本之木，效果一定会大打折扣，开发人员或相关的项目负责人对软件的质量也将很难有信心。

测试的种类很多，大致包括单元测试、集成测试、端到端自动化测试、性能测试和安全测试等。之所以需要对测试进行分类，原因是不同目的的测试对环境的需求不同，分类后可以在自动化构建中决定执行的顺序和部署条件。例如，如果每次在代码提交后立即进行自动化测试，则整个过程首先需要构建虚拟化的操作系统、安装数据库以及部署软件。考虑到自动化测试本身耗时较长而代码提交的频率很高，这样的持续测试设计只会让测试严重滞后于开发进展，甚至导致CI资源耗尽。

15.2.3 审计

大型项目对代码质量的要求往往较高，从而确保产品质量与可维护性。一般而言，代码质量的要求包括：

1. 较低的代码复杂度，避免多层嵌套的条件判断

研究表明，代码复杂度越高，出现缺陷的概率越大，而可维护性也越低。

2. 避免重复代码

对重复代码进行重构是很多项目里无法避免的课题，很容易造成对额外编写的代码增加测试成本。同时，因为代码出现在项目的多个地方，任何一次缺陷修复都需要多次查找，多次分析，维护成本很高。

3. 符合代码规范

不同的编程语言尽管语法千差万别，但都有各自的语法规范和最佳实践。例如.NET平台下的FoxCop和JavaScript常用的JSHint等。

4. 单元测试的代码覆盖率

良好的代码覆盖率能确保代码按开发人员设计的路径工作，避免自动化测试无法考虑到的边缘情况。

测试本身是动态的，需要通过测试用例生成报告，对软件规格进行描述。代码审计绝大多数是静态的，例如代码规范检查的工具JSHint可以作为单独的步骤对JavaScript代码进行检查并生成质量报告。近几年，随着JavaScript构建工具的兴盛，很多项目开始把代码

审计归并到测试概念里，审计报告也属于测试报告的一部分。另外，像代码覆盖率这种审计本身就需要由单元测试的驱动来完成，因此很自然地也成为了测试的一部分。

15.2.4　部署

持续部署能工作的软件，是CI系统里很重要的一环，良好地实施持续部署会让项目大大受益。虽然每个产品都有其自身的独特之处，但无非包括操作系统、外部依赖和产品本身，高效地创建能工作的软件环境可以让通常最容易出错的部分彻底自动化。这在减轻运维负担的同时，也让开发人员能够随时了解项目最新进展和可部署状态，不至于在发布之前才仓促加班，寻找生产环境与开发环境的区别。

为了实现持续部署，建议首先要提供干净的环境以减少对已有软件的依赖假定，另外每次构建都应该打上标签，如果出现了问题，可以迅速回滚到上个标签所在之处。

15.2.5　反馈

反馈是持续集成的输出，CI系统前期所有功能都是为了让反馈成为开发人员获取一手信息的窗口。持续反馈不会改变软件本身，但提供了手段，在特定的事件发生时，将不同的信息发送给不同的角色。收到信息的人员对信息完全理解，可以在第一时间采取最有效的措施来解决问题。

对于开发团队而言，能够基于测试结果了解最新版本的工作情况，决定是否进行回滚或缺陷修复，这些信息也可以汇集起来确定项目的发展趋势。除开发团队外，客户、项目经理和投资人也可以通过反馈渠道沟通信息，而沟通是否准确和及时正是由反馈方式所决定的，这些方式包括邮件、声音、电话或者可视设备等。

15.3　如何实施持续集成

15.3.1　消除误解

尽管持续集成有诸多好处，很多项目团队却对其望而却步，原因是诸多误解让开发人

员以为持续集成会是一件工程浩大却吃力不讨好的事情，所以在实施持续集成之前首先要消除项目团队的误解：

1. 增加了运维成本

实际上，无论是否使用了持续集成，CI系统中提到的5大功能点（编译、测试、审计、部署和反馈）都必不可少，无非是这些步骤通过手工过程控制还是使用CI系统代为管理而已。与其使用不可控的手工控制，不如管理一个健壮的CI系统，其带来的便利性不言而喻。

2. 害怕失败的反馈

复杂项目的开发不可能一路坦途，持续集成的核心精神正是经常地提交代码，让失败的反馈使开发人员知道错误发生在哪个变更或者哪个版本上，这才是高质量控制的途径。如果因为害怕收到失败的反馈，干脆做一只埋起头来的鸵鸟，但只会让所有问题在后期全面暴露，影响整个产品的正常发布，甚至导致项目失败。

3. CI系统过于复杂

恰恰相反，作为CI系统基石的构建工具已经是非常成熟的技术，Ant、NAnt、make、MSBuild、Rake、gulp等都可以轻松满足各种自动化需求，结合CI工具提供的工作流，CI系统的部署和实施不仅不再是一个复杂无趣的业务，反而是一个充满乐趣和鼓励创新的前沿技术。

15.3.2 前提条件

要想成功实施持续集成，需要开发团队能够完全遵守实践的准则，前提条件包括以下几点：

1. 版本控制库

一般来说，对于团队型开发，只要人数超过两个，就很难保证所有人员能够互不干扰的变更代码了。为了更高效地工作，与项目相关的所有内容都应该提交到版本控制库，包括产品代码、测试代码、构建脚本以及外部依赖等。

版本控制库通过受控的代码仓库管理所有与软件开发相关的资产变更，为团队成员提供权威的代码访问渠道。沿着时间回溯，开发人员可以取得文件的不同版本。基于版本控制库，团队成员既能同时工作于软件的不同组件，又能维护之间的协作以及提交记录。版本控制工具种类众多，比较出名的有Perforce、ClearCase、CVS、Subversion、TFS和Git等。

2. 自动化构建工具

自动化构建工具一般至少要包括代码编译、组件打包、程序执行等功能。编译源代码是构建的主要工作之一，为了提高效率，编译应该根据相应的源代码是否发生改变而有条件地执行。组件打包是将编译的结果和其他需要包含的文件等组织在一起，形成可以部署的组件。构建工具应该知道何时需要重新打包。程序执行是指构建工具在它支持的平台上，调用所有提供命令行接口的程序。常用的构建工具包括Ant、NAnt、MSBuild、make、Maven、Rake和gulp等，开发人员应该将构建脚本和代码同等对待，并不断重构和优化，以保证其整洁和易于理解。

3. 团队合作与共识

持续集成是一种开发实践而不是一个死板的工具。由于其本身囊括了软件开发生命周期的方方面面，开发人员、测试人员、项目经理乃至运维人员都投身其中，良好的团队合作是实施持续集成的基础。

另外，它需要团队给予一定的投入并遵守约定的准则，需要每个开发人员都能以小步增量的方式频繁地修改代码并进行提交。如果大家不能按准则实施，例如长期在本地进行私有构建而不进行提交，则持续集成系统无法对其进行检验，更无法如期望的那样通过持续集成提高软件的质量。

15.3.3 CI工具

实施持续集成有多种选择，可以创建自己的集成工具，也可以手工执行，但大型团队往往使用多种技术开发，每天可能会积累很多代码变更，自己创建的工具很难完全满足所有需求并提供高品质的集成效果。

编译、测试和部署等工作虽然可以通过脚本来集成，但CI工具能够提供更直接的集成方式，效率更高。现在市场上有许多优秀的CI工具，例如Bamboo、Travis CI、TeamCity、Buddy、Go和Jenkins等，其中甚至不乏免费的产品。这些CI工具在安装后，仅需要几分钟即可完成基本配置，实施起来非常方便。同时，它们配置好后即原生提供集成需要的功能，并能够很容易地进行扩展。因此，在实施持续集成的时候建议开发团队使用已有的CI工具，而没必要自己专门写一个。

从根本上而言，CI工具就是一个自动化的交付流程。但这并不意味着使用了CI工具就不需要人工参与了，而是把执行过程中那些容易出错且复杂的步骤通过可靠的工具来持续地自动完成。在这个过程中，开发人员从单调易出错的脚本编译等工作和流程中解脱出

来，以便专注于更有价值、更有创造力的业务设计、缺陷调试、沟通协调等工作中。

CI工具一般由两部分组成。一部分是一个长期运行的工作流，以特定的时间间隔轮询版本控制库中的变更，定期或基于事件触发某些构建，向相应人员发送反馈等。某些CI工具还包括扩展功能，例如执行开发者测试、文档集成、部署功能和代码品质分析等。另一部分一般以Web服务器的形式存在，通过信息面板显示构建历史，查看执行结果是成功还是失败，以及具体的详情报告等。

图15-1展现了一个常见的CI场景：

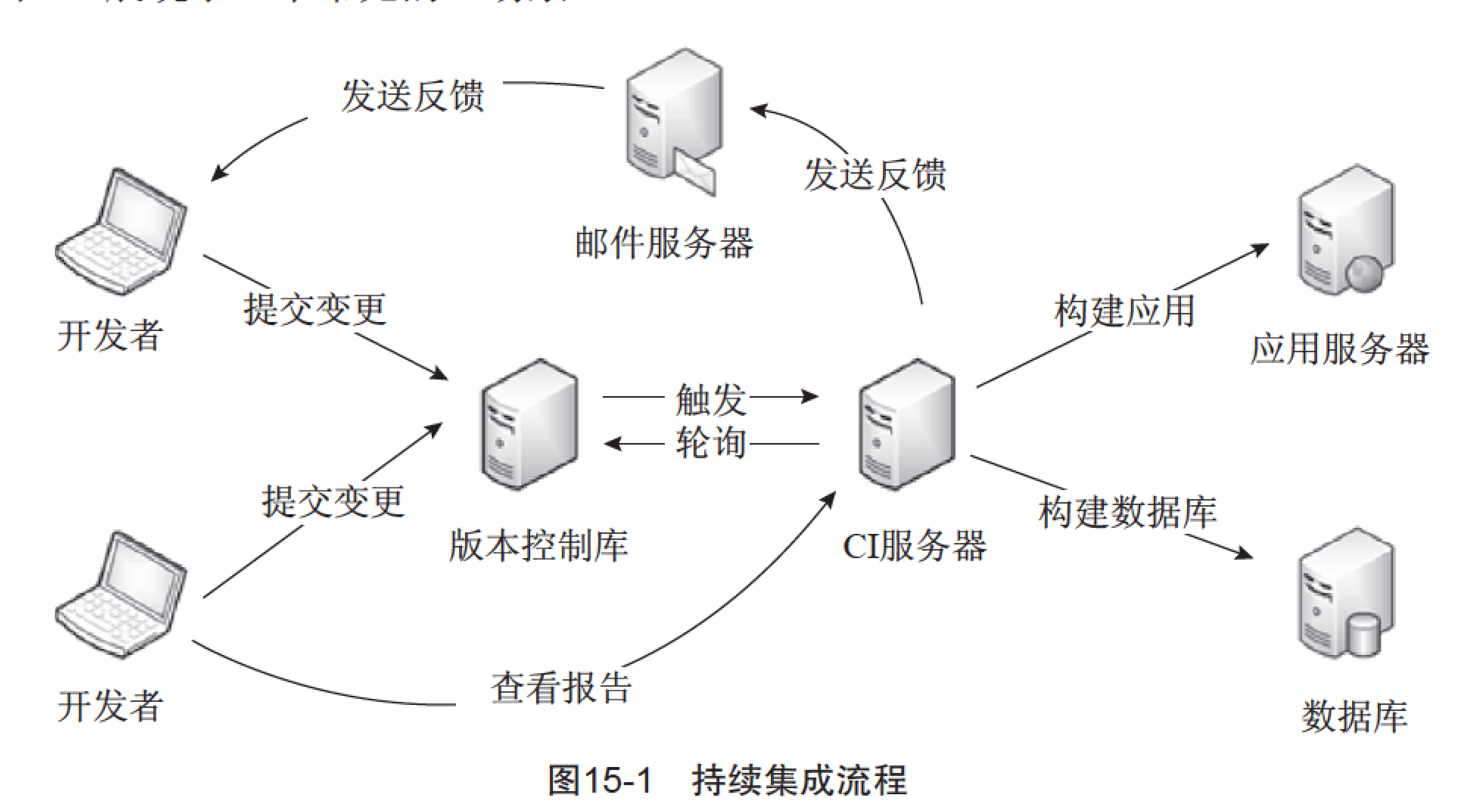


图15-1　持续集成流程

（1）开发者修改代码进行本地构建，确认无误后，向版本控制库提交变更。

（2）代码提交后，CI服务器通过轮询检查到新的变更；或者版本控制库可以触发事件通知CI服务器发生了提交变更。

（3）CI服务器取得最新的代码副本执行构建，包括构建应用、构建数据库和执行测试等。

（4）CI服务器向开发人员发送邮件反馈；开发人员也可以通过访问CI服务器的Web站点查询构建结果。

15.3.4　实践准则

实施持续集成遵循以下实践准则：

1. 专门的CI服务器

CI服务器是持续集成的核心部件，用于执行构建脚本。建议使用专门的服务器运行CI工具，从而保持其环境的干净，避免不必要的冲突导致的构建失败。

2. 频繁地提交代码

只有频繁地提交变更，才能享受到版本控制所带来的诸多好处，比如能轻松回滚到某个之前的版本。另外，一旦变更提交到版本控制库，开发团队的其他成员就能够立刻看到并及时与之同步。反之，如果把若干天的代码修改作为一次变更进行提交将会遇到较多的变更冲突，需要更多的时间进行集成。

3. 提交前执行本地构建

为了防止集成构建失败，开发人员应该在提交前在自己本地环境内执行一次构建，这将确保有缺陷的变更不会导致集成环境里的错误。当然，这也取决于具体情况，诸如自动化测试等耗时较长或需要部署环境的测试，可以考虑在本地省略或只执行冒烟测试。

4. 每次变更触发构建

软件可以分为代码、配置信息和数据，其中任何一部分发生了变化都可能导致软件行为发生变化。所以，每次变更提交后都应该触发构建，特别是单元测试和代码审计，确保所有的修改都得到验证。

5. 快速构建

如果整个构建包括编译、测试和审计需要花很长时间执行的话，持续集成的效果会打折扣。一方面，如果构建时间较长，则每次构建也会相应包含较多的变更，若出现错误，很难确定是哪个变更造成的。另外，代码提交或者集成的效率也会降低，因为开发人员需要等本地构建完成并确认当前的最新版本可以成功构建后再提交代码。

如果项目庞大，构建速度无法完全优化，则建议考虑使用分布式的CI架构。

6. 只生成一次二进制文件

很多构建系统会在同步版本控制库中的源代码后，在不同的上下文中重复执行编译，甚至在不同环境中部署前都要重新编译一次。但对于同一份源代码，每次都重新编译会引入结果不一致的风险，有可能引起软件的行为不一致现象。

7. 尽快接受反馈

开发人员能够对反馈在第一时间作出反应，例如当出现构建失败，可以立即展开修复工作是持续集成的基础。否则，如果构建失败的邮件已经发出，而开发人员却没有查看邮件，那么这个构建错误可能会长期在版本控制库里存在，直接影响到后续版本的构建。对于这种情况，除了邮件反馈，还可以考虑其他更及时的反馈方式，例如短消息或电话等，在非工作时间也可以安排值班的开发人员专门解决构建错误。

8. 优先修复导致构建失败的缺陷

无法集成的构建是失败的构建，可能存在编译错误，测试或审计失败，也有可能是部

署问题。在CI环境下，这些问题需要优先修复，否则后续的所有版本都将无法构建，特别是编译和部署问题会直接导致无法生成测试报告，使项目经理无法跟踪当前的项目进展。

实际上，频繁的以增量方式提交变更，出错后立刻调查是最容易也是最节省时间的实践。

15.4 选择持续集成工具

目前市场上成熟的CI工具很多，包括商业的和开源的。在选择CI工具的时候，并不一定要找最好的、最贵的，每个工具都有自己的优点和不足，因此最重要的是找到最适合自己项目的工具。

在进行选择决策的时候，如果公司内已经有了正在运行的CI案例，建议先参考现有案例的使用经验和反馈，结合新项目的需求看其是否符合要求。一般而言，如果需求基本符合，建议使用现有的CI工具，无论是使用经验还是排错方面，都会大大节省成本。

如果考虑选择新的CI工具，可以从以下几个方面权衡：

1. 功能

CI工具的功能是优先需要考虑的方面，它关系到对项目的直接支持水平，例如处理构建的能力、反馈报告的种类等。

2. 可靠性

如果一个CI工具功能很强大，却频频引发意外死机，无疑不是用户希望看到的结果，因而考量CI工具的可靠性也非常重要。一般而言，用户数量是衡量工具软件是否优秀的参照指标。一个工具软件如果下载或使用的人数较多，往往说明其比较稳定。

3. 平台

一个CI工具并不一定能够支持所有的平台，有些工具可以运行在Windows上，而有些只能运行在Linux上，需要结合软硬件来看其是否符合本公司的使用习惯与资源条件。

4. 易用性

良好的易用性可以节省大量的部署与配置时间，也可以节省使用者的学习时间。

5. 可扩展性

项目不是一成不变的，可能开始只需要用JavaScript和C#来编程，随着需求的变更，后面又用到了其他默认不支持的编程语言。如果CI工具有良好的扩展性，能够通过插件或

脚本解决此问题，无疑会方便很多。

6. 版本控制工具的支持

CI工具需要支持项目里用到的版本控制库，否则集成无法进行。表15-1列出了一些常用CI工具对主流版本控制库的支持情况。虽然Travis CI和CircleCI当前只支持Git，但它们是基于云服务的持续集成工具，用户无需自己维护CI工具的部署，因此目前也非常流行。

表15-1　常用CI工具对主流版本控制库的支持情况

	Perforce	ClearCase	CVS	Subversion	TFS	Git
Jenkins	是	是	是	是	是	是
Travis CI	否	否	否	否	否	是
CircleCI	否	否	否	否	否	是
TeamCity	是	是	是	是	是	是
Bamboo	是	是	是	是	是	是

第16章 持续测试

持续测试是CI系统的核心功能，也是高质量发布的保证。在一个功能完善的CI系统中，持续测试必不可少。单元测试和自动化测试虽然目的都是检验软件的实现质量，但也有各自的独特性，本章将基于已介绍的单元测试和自动化测试的知识，介绍如何在CI系统中实施持续测试。

本章将介绍：

- 测试策略
- 基于Jenkins的持续集成
- 集成Team Foundation Server
- 集成Visual Studio Team Services
- 集成Github

16.1 测试策略

随着敏捷开发和 DevOps 实践被越来越多地采用，在每次迭代中手动触发单元测试和自动化测试已经成为不可持续的模式。大量时间消耗在等待构建上、缺乏足够的灵活性，都会降低测试回报。特别是测试反馈速度太慢会显著降低生产力，而较高的测试效率恰恰是跟上快节奏的开发生命周期的一个关键。这时候就需要引入持续测试。

通过更敏捷的测试尽可能地找到更多的问题，可以改善测试效率。持续测试通过提供实时测试和及时反馈，能够加强整个团队的信任度。对于项目利益相关者，持续测试提高了对成功交付的信心。对于开发团队，持续测试最小化了测试的影响范围，这可能能够减少开发成本，实现更快的项目交付，更重要的是，确保了高质量、可靠的解决方案。

要充分发挥持续测试的价值，关键是在开发的生命周期中更早、更频繁地进行测试。这样，团队就可以尽早地测试风险最大的元素，然后不断重用这些测试用例。测试的同时，尽早向开发团队提供代码质量的迭代式反馈，可以加快修复缺陷的速度，确保在开发生命周期的后期发现的问题更少。特别是单元测试，如果能够频繁地进行，则每个测试版本之间的差异会很小，大大减少与发布相关的风险。为了做到尽早、更频繁地进行单元测试，任何一次代码变更都应该触发构建和反馈流程，单元测试一般运行速度很快，可以让开发人员尽早得到测试结果。

为了提高单元测试和自动化测试的效率，对于没有相互影响的测试用例可以使用并行的方式驱动，前面章节介绍的Karma和Protractor都能够支持并行测试。

与单元测试不同，自动化测试需要将产品部署后模拟用户对其操作进行功能上的检查。无论是部署还是自动化测试本身，一般都耗时较长，反馈时间相应会比较晚。所以，在构建时应该先执行较快的测试即单元测试，保证开发人员尽快得到反馈。另外，自动化测试不仅构建时间长，对测试资源（例如Selenium Server）的消耗也更多，即便有些公司使用了虚拟化技术实时创建虚拟机作为测试环境，仍然可能存在资源不够的情况。所以，区别于单元测试，不建议每次代码变更都触发自动化测试，而可以考虑在每个新版本发布后进行自动化测试或在每天的一个固定时间同步最新代码进行每日构建，这也叫作Daily Build。

基于Protractor的自动化测试不建议直接运行在构建工具或CI服务器上，因为在同一台机器上安装多种浏览器不仅不能提高测试效率，反而会让CI服务器功能发散，难以维护。很难想象每次浏览器出了新版本后还要到各个CI服务器上进行更新。所以，集成环境内的自动化测试建议考虑基于Selenium Grid或Sauce Labs等云服务通过分布式测试进行集成。

另外，自动化测试不可能覆盖到所有的用户使用情况，也无法测试经由系统内的所有路径，尝试这么做的成本只会最后大大超过收益。所以，无论是从编写测试用例的成本而言，还是从测试的效率考虑，都应该让自动化测试用例尽可能多地覆盖最重要和最常用的使用场景，而不是刻意地面面俱到。这样才能让持续集成的效果更好，价值更高。

16.2　基于Jenkins的持续集成

在众多的CI工具中，Jenkins因其完善的功能和强大的社区支持，得到了广泛的关注和

使用，是当前最流行的CI工具之一。

Jenkins的前身是Hudson，由当时还在Sun的Kohsuke Kawaguchi及其团队进行研发。在Oracle收购Sun之后，开源社区于2011年1月通过投票，决定将Hudson项目更名为Jenkins。

以下为Jenkins的主要特点：

- 免费而且开源。Jenkins是一个完全免费的CI工具，其源代码完全公开于GitHub。
- 丰富的学习资料。Jenkins因为其开放的特性得到了社区的大力支持，可以很方便地查找到需要的学习资料。
- 容易安装与配置，无需专门的数据库即可运行。
- 支持分布式构建。通过Master和Slave模式，Jenkins可以把构建任务轻松地分发到多个Slave上。
- 跨平台。Jenkins几乎支持所有的平台，包括Windows、Ubuntu、Fedora和CentOS等。
- 可读的永久链接。Jenkins对于大部分页面都可生成永久链接，方便其他地方进行引用。
- 良好的扩展性。Jenkins拥有强大的插件框架，通过大量的插件，可以支持几乎所有技术类型的持续集成。
- 稳定的产品线。Jenkins的产品线稳定，通过大概每三个月发布的Long-Term Support（LTS）版本提供最稳定的功能。

Jenkins是一个基于Java开发的工具，在使用前请先参考12.1.1节安装Java JDK。Jenkins的安装非常简单，从网址https://jenkins.io/index.html处下载jenkins.war并保存到服务器后，即可在命令控制台执行以下命令启动Jenkins服务（参见图16-1）。

```
java -jar jenkins.war -httpPort=8080
```

Jenkins也支持HTTPS，可以使用参数httpsPort，并提供证书进行配置①。另外，在Windows平台，Jenkins也可以通过传统的安装包将其安装为Windows服务，这样每次机器重启后Jenkins也会自动启动。

图16-2为启动后的Jenkins主界面，左侧的导航栏是Jenkins主要功能的入口，例如New Item用于创建Jenkins构建项，People定义了Jenkins内的账号，Build History用于查看构建历史等；右侧是已经创建好的构建项。

① Jenkins. Starting and Accessing Jenkins[OL]. [2016]. https://wiki.jenkins-ci.org/display/JENKINS/Starting+and+Accessing+Jenkins.

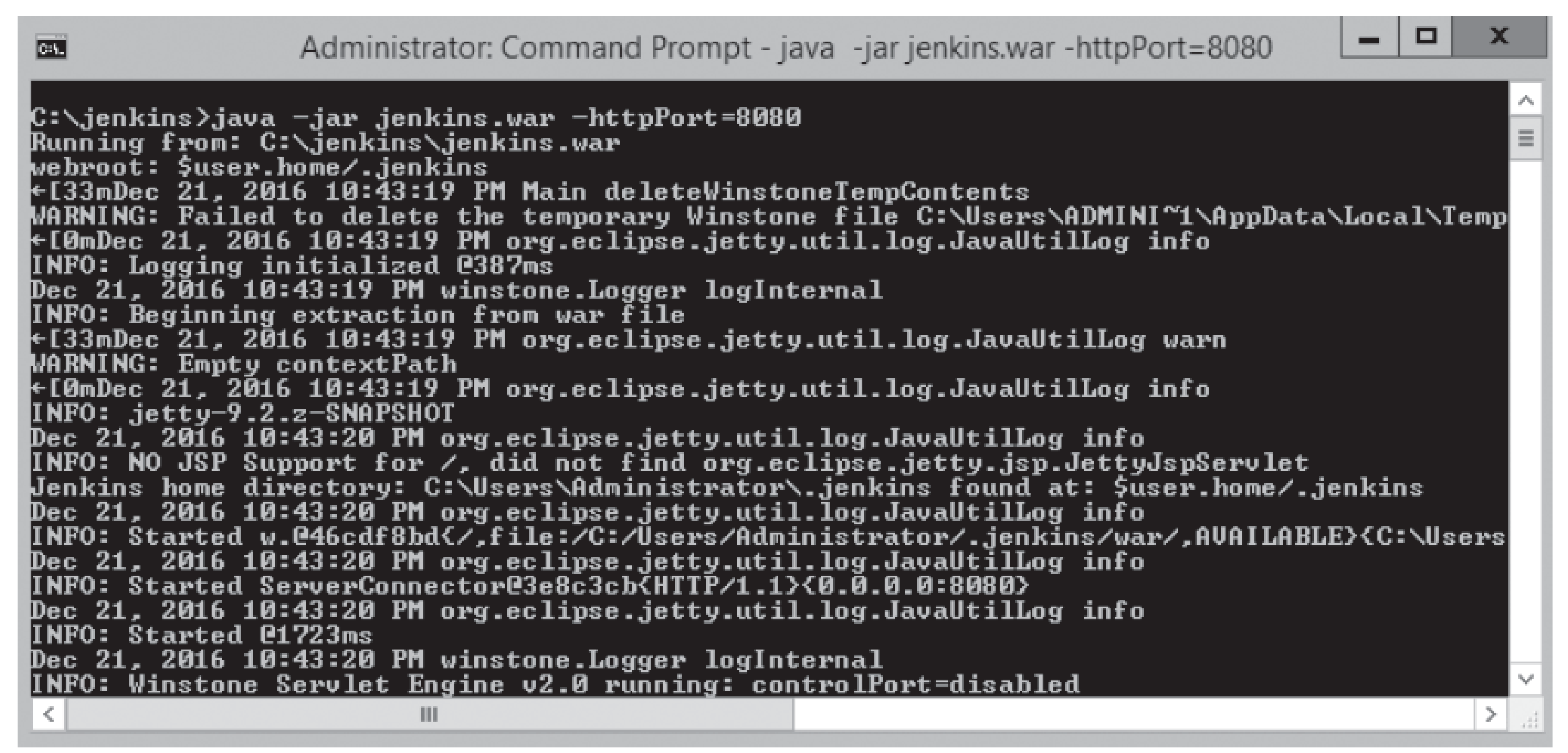


图16-1　启动Jenkins

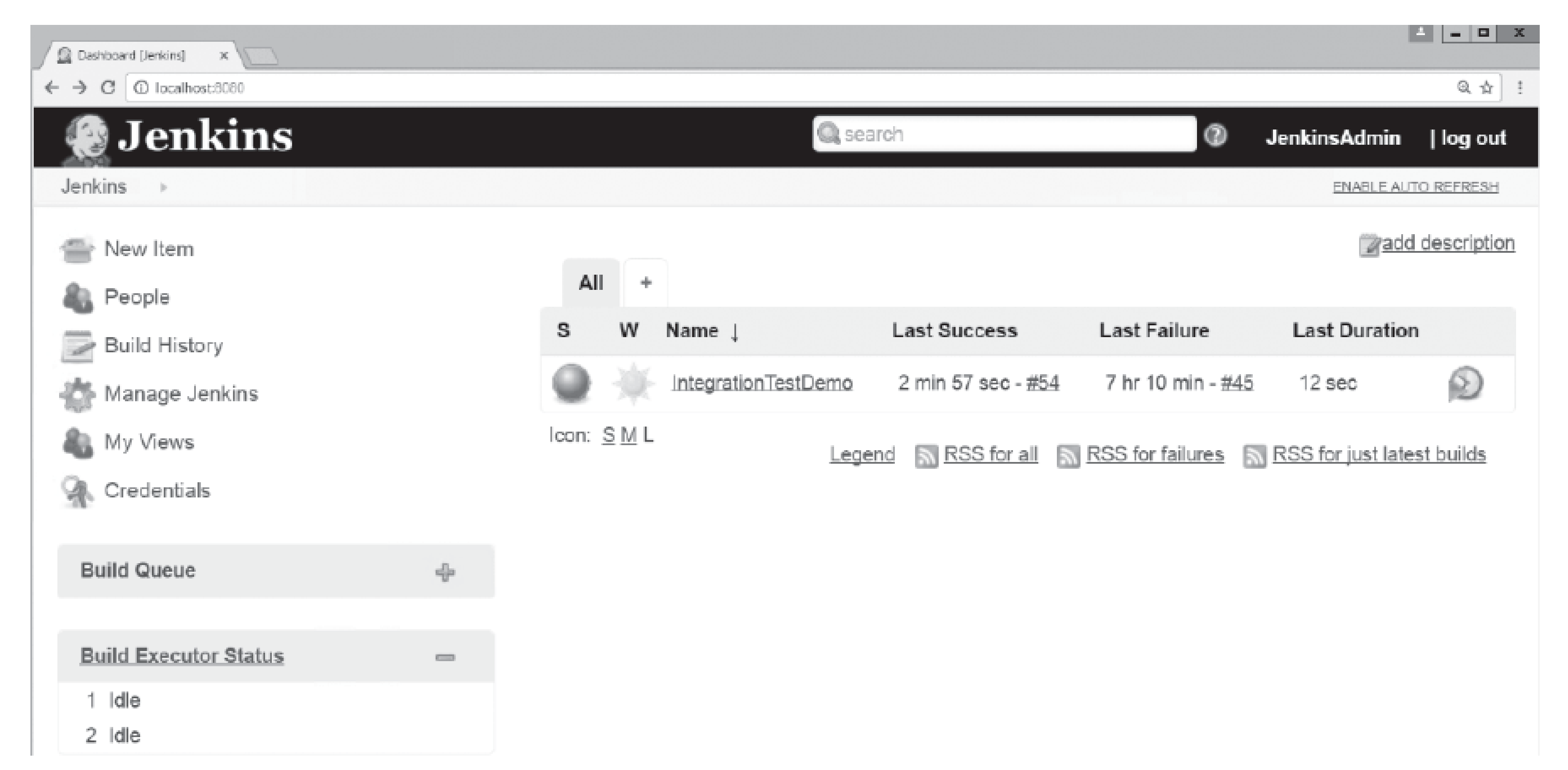


图16-2　Jenkins主界面

对Jenkins的绝对大多管理工作都是通过Manage Jenkins导航栏进入的，如图16-3所示。其中，Configure System项用于配置全局内有效的设置，Global Tool Configuration用于配置集成所用工具的路径和安装，Manage Plugins用于安装卸载Jenkins插件等。

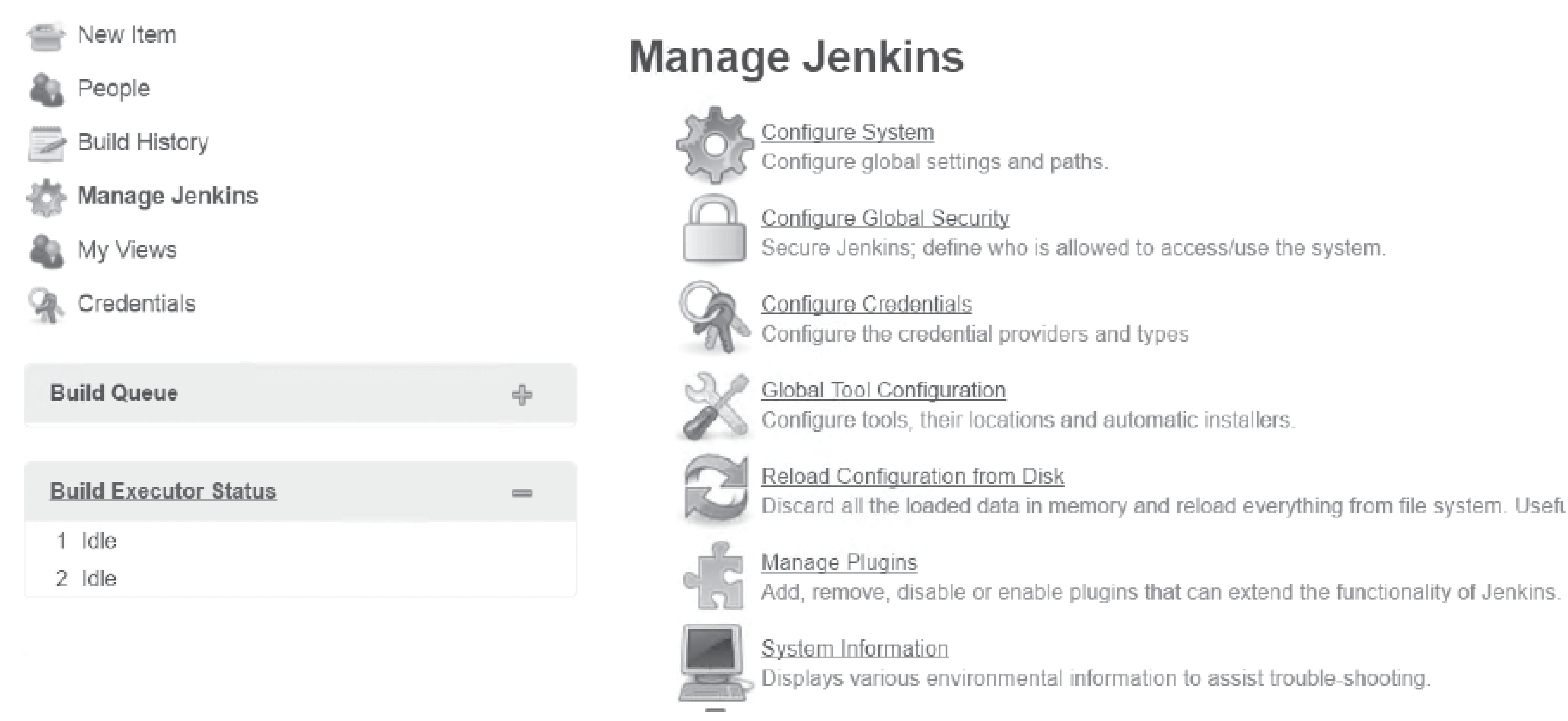


图16-3　管理Jenkins

在下一节笔者将介绍配置Jenkins的方法，为了让其与TFS的Git代码仓库同步代码，所以需要先安装Git并在Global Tool Configuration中配置Git的安装路径，如图16-4所示。Git安装包可以从网址https://git-scm.com/downloads处下载得到。

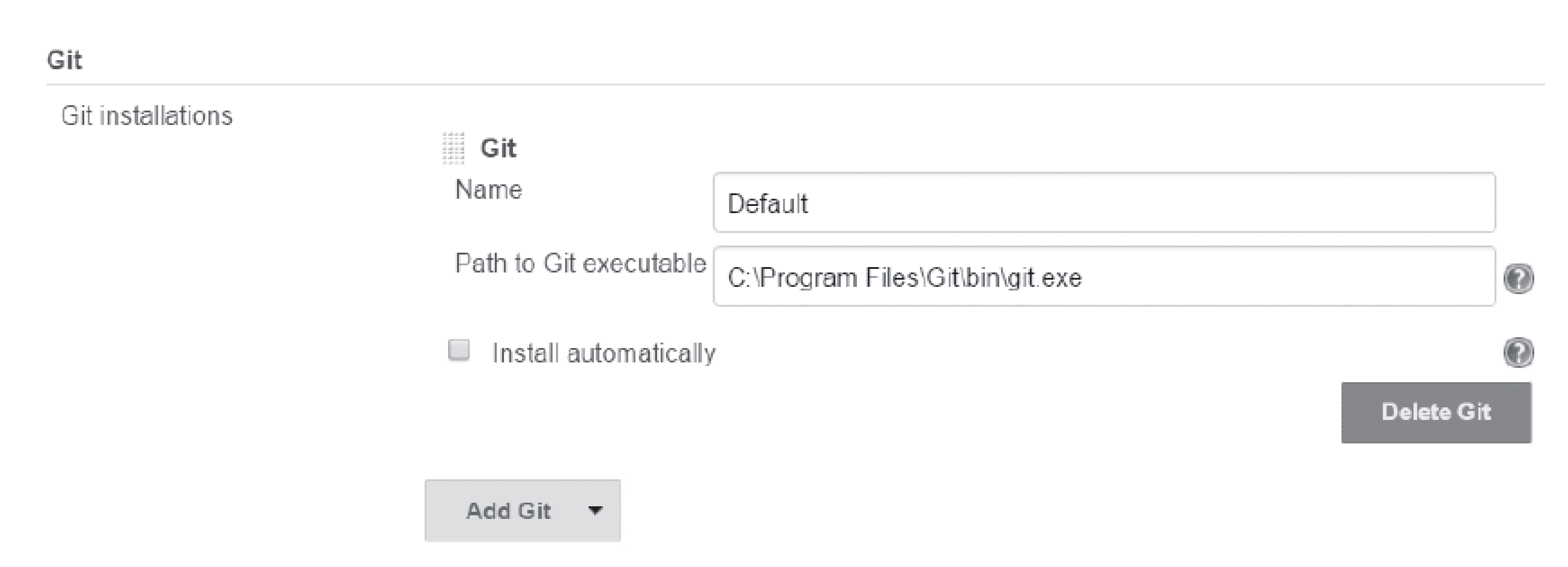


图16-4　配置Git

在后续章节中，将进一步以Jenkins为例展示如何通过持续集成驱动单元测试和自动化测试。持续集成的其他功能包括软件和数据库的部署等，基于不同的技术与产品，其构建方式也相应不同，如果读者有兴趣，建议参考专门的部署书籍进一步学习。

16.3　集成Team Foundation Server

Team Foundation Server（TFS）是微软公司发布的软件开发生命周期管理套件，除了版本控制和源代码管理外，TFS还具有工作项跟踪、产品发布和集成SharePoint等功能，是Windows平台上使用最广泛的开发管理工具之一。TFS 的数据仓库基于SQL Server建立，存储工作项跟踪、源代码管理、版本和测试工具的数据。

TFS项目管理、配置等工作可以通过Web管理页面进行，功能强大，简单易学，本书后续演示基于TFS 2015 Update 3进行。尽管将TFS安装到Windows域环境内对用户管理会方便很多，但这并不是必须条件。

16.3.1　创建项目

使用TFS创建项目的步骤如下：

（1）TFS的项目管理界面是一个运行在浏览器环境下的Web应用，单击New team project项创建一个新项目。TFS支持Team Foundation Version Control和Git两种版本管理的

方式。如图16-5所示，选择Git选项创建项目TestProject。

Control panel > DefaultCollection
Overview　Security　Build　Agent queues　Extensions*
Not all collection level administration is exposed in the web experience. For all administration operations at the collection level please use the Administration Console on your Team Foundation Server.
Collection profile　Projects
Name
DefaultCollection
Description
New team project
CREATE NEW TEAM PROJECT
Project name　TestProject
Description
Process template　Agile
This template is flexible and will work great for most teams using Agile planning methods, including those practicing Scrum.
Version control　Git
Team Foundation Version Control
Git
repository.
Create project　Cancel

图16-5　创建新项目

（2）创建好项目后，即可得到其访问链接，如图16-6所示。

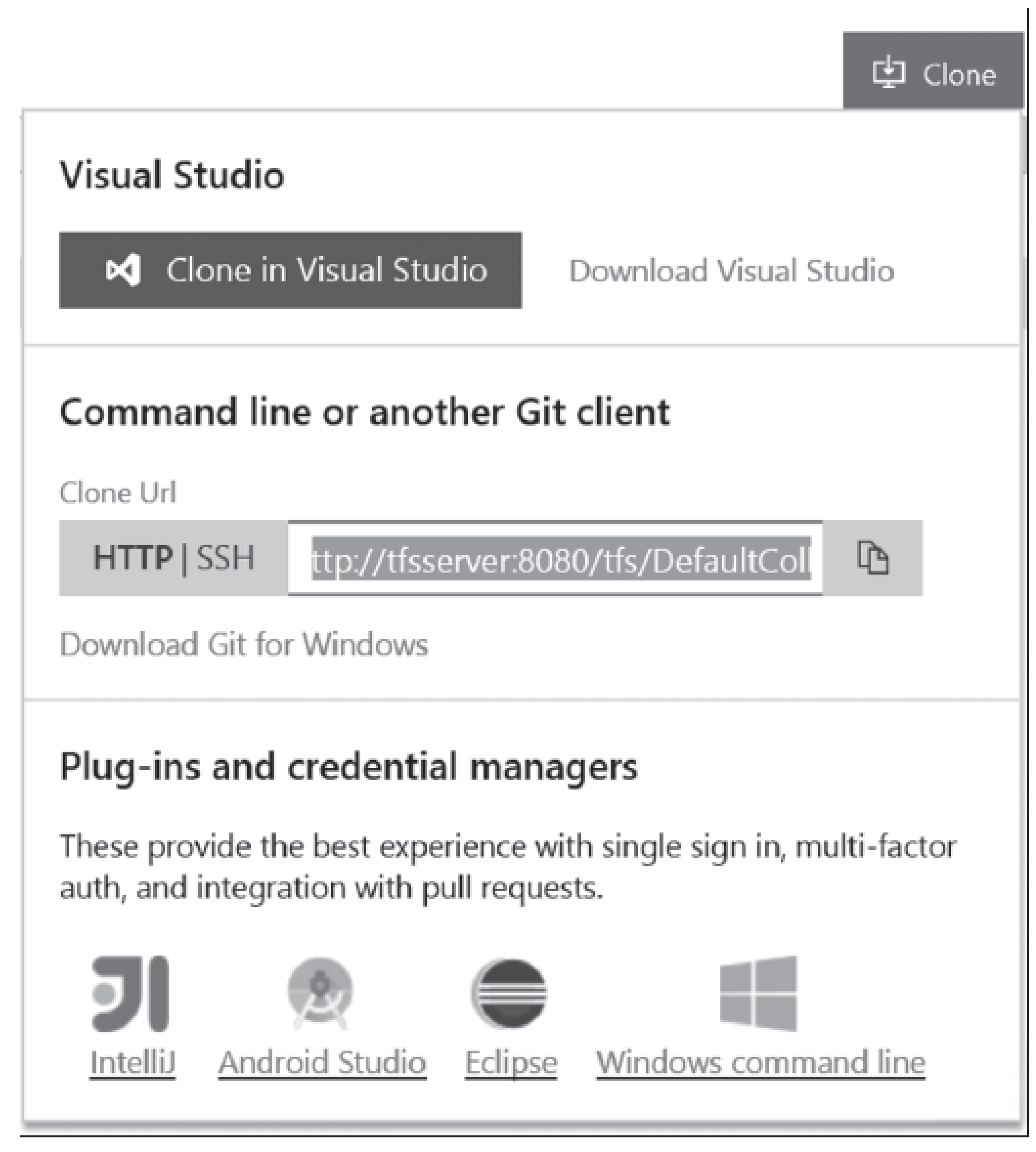


图16-6　项目的Git访问链接

16.3.2 从Visual Studio Code提交变更

TFS下的Git版本控制实际是一个基于HTTP的服务，包括Visual Studio等任何支持Git的IDE均可以访问使用。从Visual Studio Code提交变更的步骤如下：

（1）如果使用Visual Studio Code作为项目的客户端编辑工具，需要先通过上节获得的Git访问链接将项目克隆到本地，如图16-7所示。同时，需要使用git config配置用户名和邮件。

```
C:\repositories>git clone http://tfsserver:8080/tfs/DefaultCollection/TestProject/_git/TestProject
Cloning into 'TestProject'...
remote:
remote:                   fTfs
remote:                 fSSSSSSSs
remote:               fSSSSSSSSSS
remote: TSSf        fSSSSSSSSSSSS
remote: SSSSSF    fSSSSSSST SSSSS
remote: SSfSSSSSsfSSSSSSSt   SSSSS
remote: SS  tSSSSSSSSSs      SSSSS
remote: SS   fSSSSSSST       SSSSS
remote: SS fSSSSSFSSSSSSf    SSSSS
remote: SSSSSST    FSSSSSSFt SSSSS
remote: SSSSt       FSSSSSSSSSSSS
remote:               FSSSSSSSSSS
remote:                 FSSSSSSs
remote:                   FSFs    (TM)
remote:
remote:  Microsoft (R) Visual Studio (R) Team Foundation Server
remote:
Unpacking objects: 100% (54/54), done.

C:\repositories>cd TestProject

C:\repositories\TestProject>code .

C:\repositories\TestProject>git config user.name "TestUser"

C:\repositories\TestProject>git config user.email "TestUser@contoso.com"
```

图16-7　克隆项目到本地

（2）在客户端完成代码编辑后，单击Commit All按钮将修改提交到本地Git，如图16-8所示。

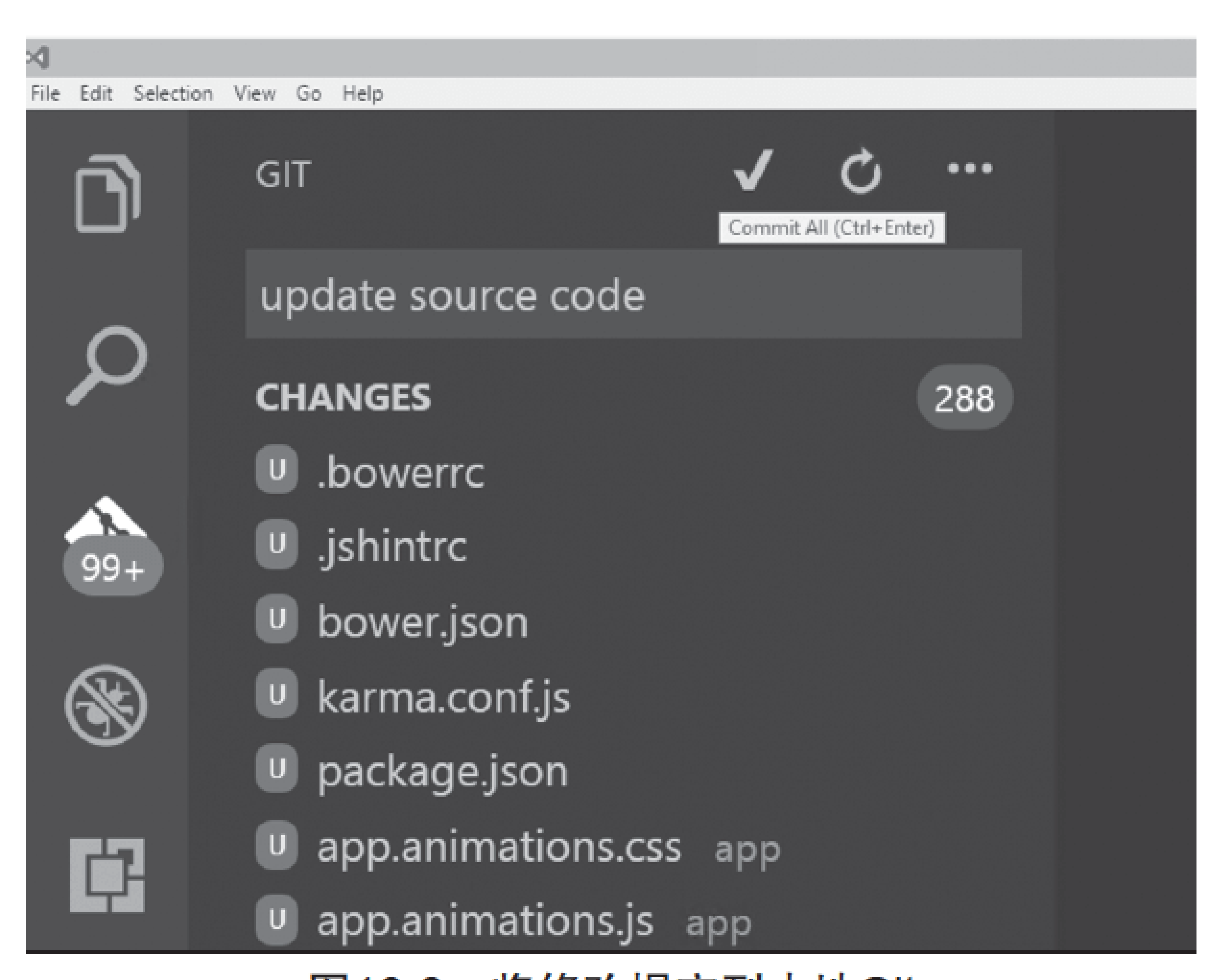


图16-8　将修改提交到本地Git

（3）如图16-9所示，单击…按钮，选择Push命令把本地的变更同步到TFS的Git代码仓库。

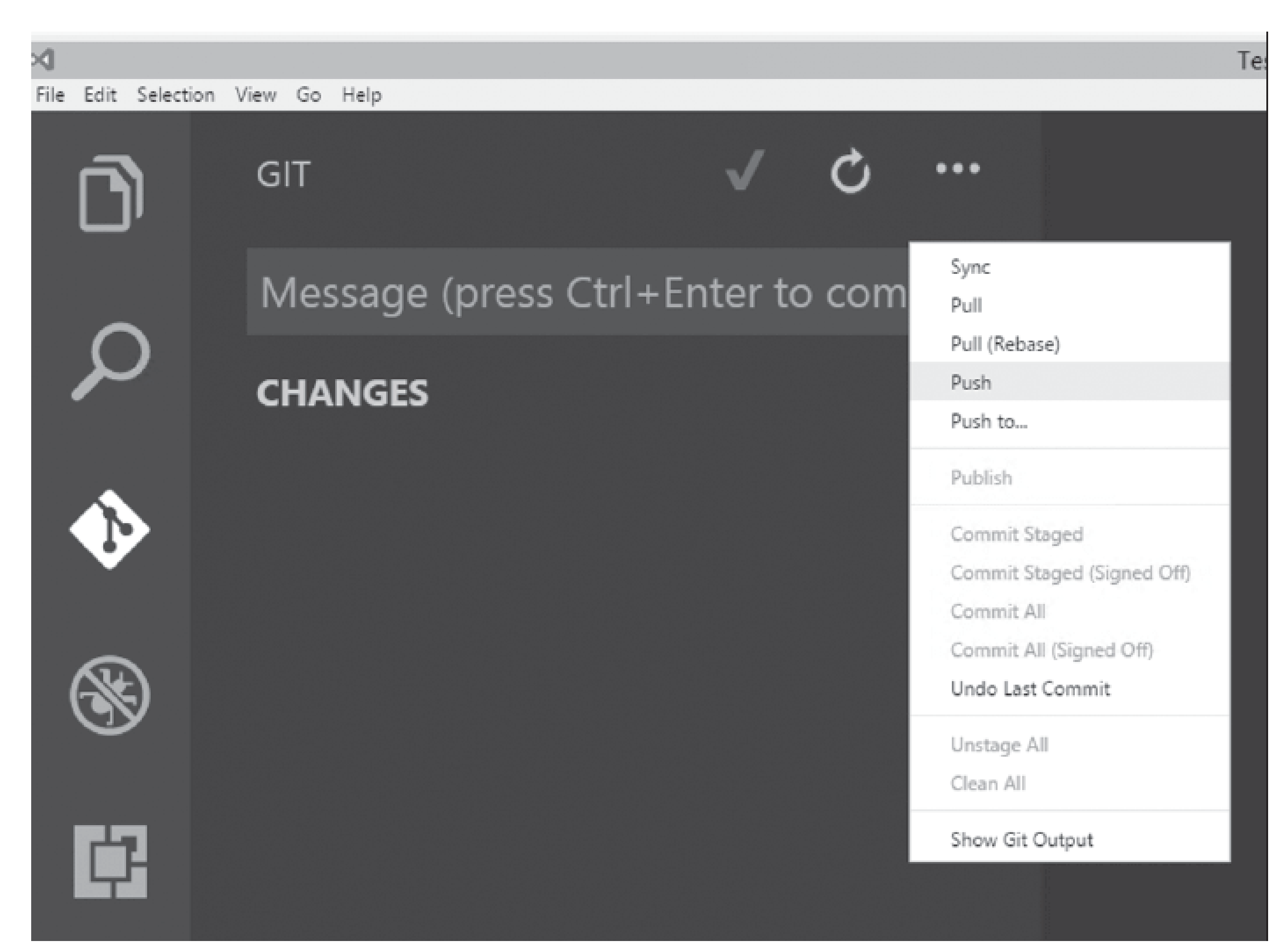


图16-9 推送到TFS

16.3.3 配置TFS插件

Jenkins之所以强大，源于其优秀的可扩展性和对大量插件的支持。配置TFS插件的步骤如下：

（1）为了将Jenkins与TFS集成起来，选择Manage Plugins选项进入插件管理界面，这里列出了所有可供下载的、已安装的和可以更新的插件。如图16-10所示，在可供下载的页面中搜索并安装Team Foundation Server Plug-in插件。

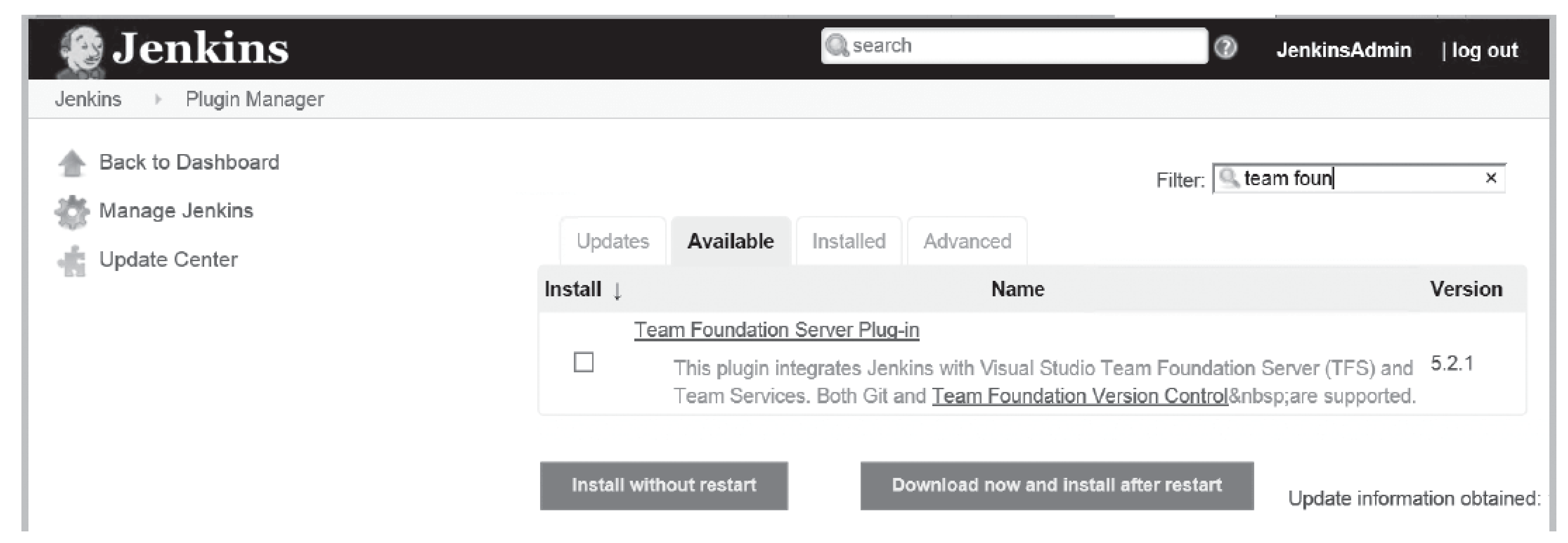


图16-10 安装TFS插件

（2）在Configure System内配置有已经安装好的TFS插件，包括工作集合的地址以及Jenkins用来访问的账户，如图16-11所示。

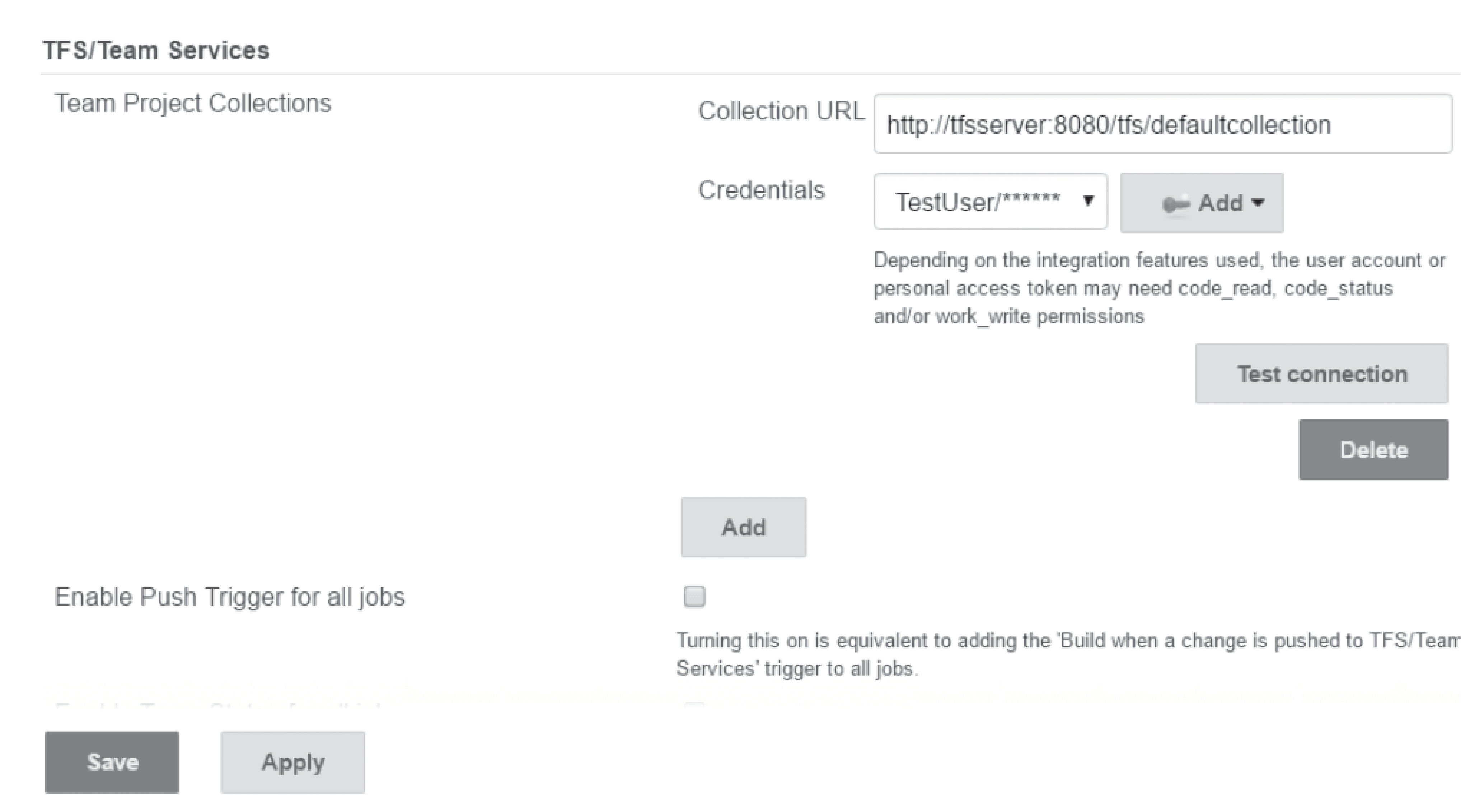


图16-11　配置TFS插件

16.3.4　创建并配置Jenkins构建项

Jenkins的集成工作是通过构建项进行的，创造及配置Jenkins构建项的步骤如下：

（1）在Jenkins主页选择New Item选项，可以看到有多种构建类型供选择，包括Freestyle project、Pipeline和External Job等。如图16-12所示，选择Freestyle project选项并将构建项命名为UnitTestDemo。该选项也是Jenkins里使用最普遍的构建选项。

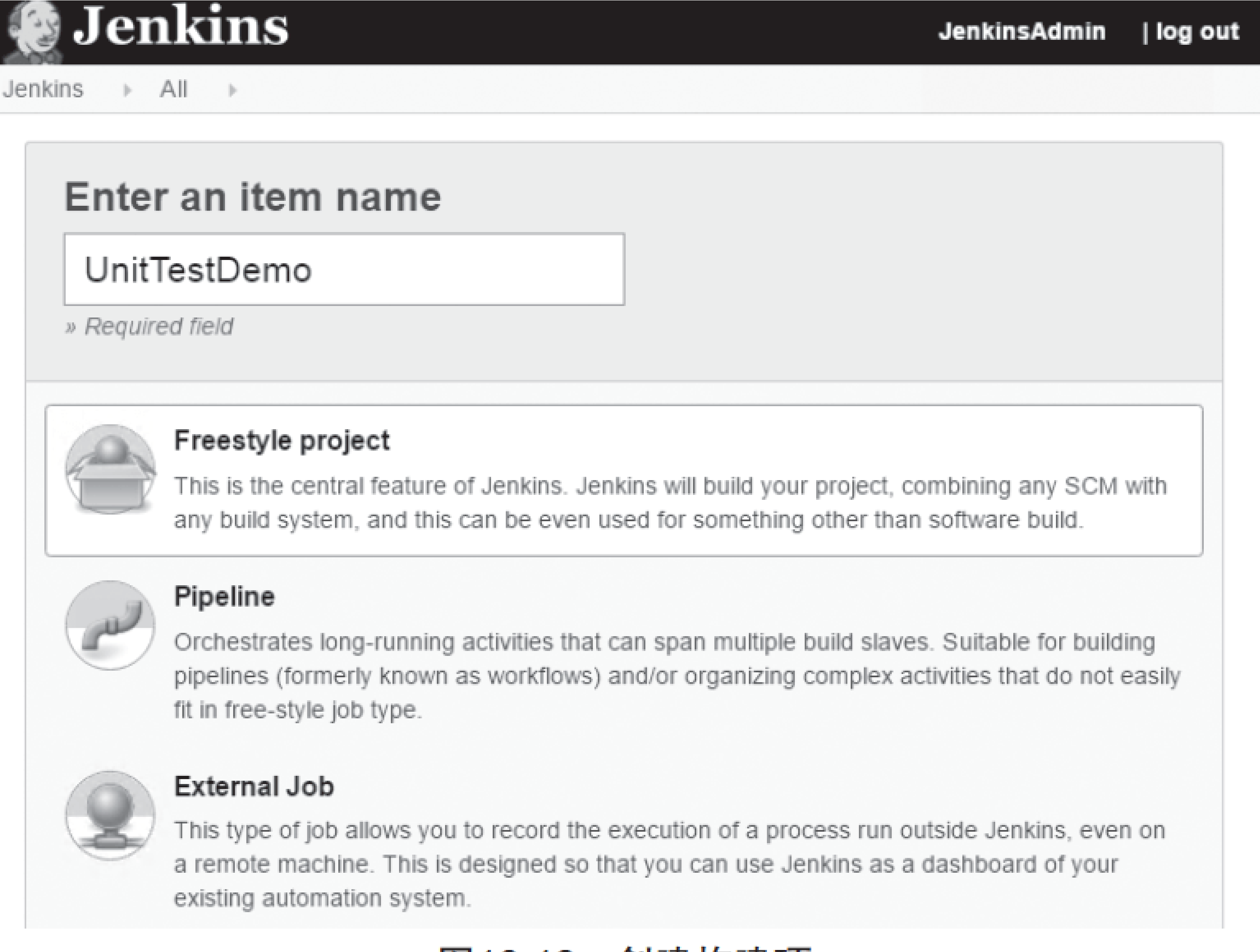


图16-12　创建构建项

（2）在TFS服务器中打开项目管理界面，该界面用于配置所有与项目相关的工作，如图16-13所示。

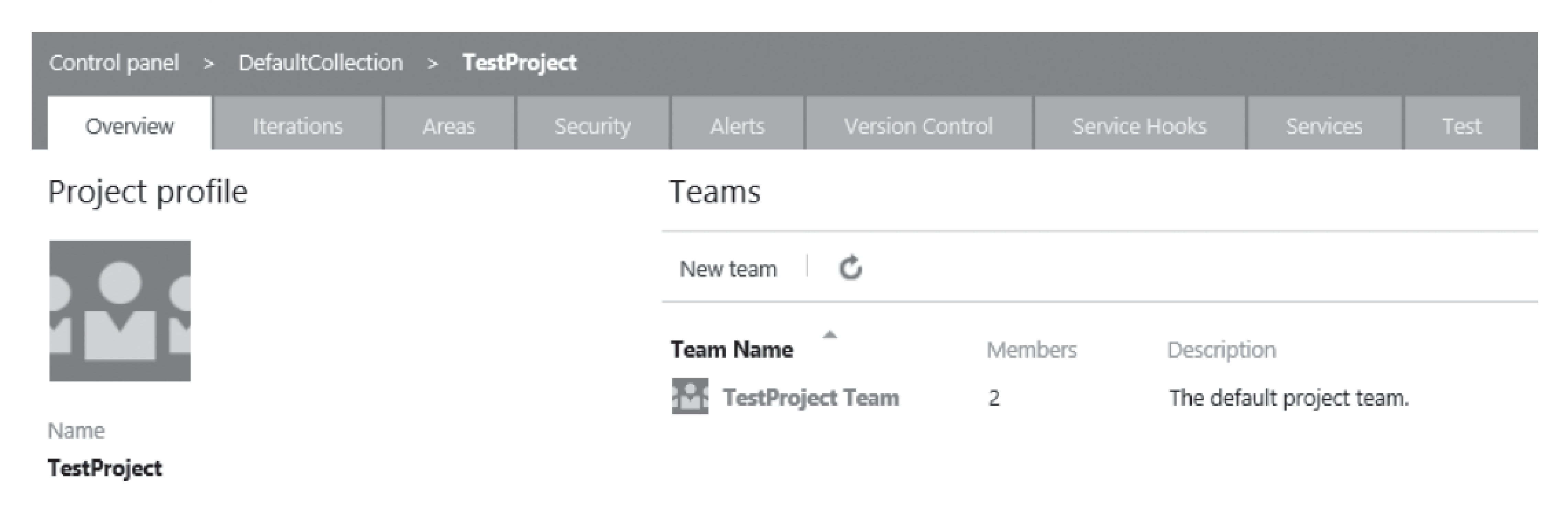


图16-13　TFS项目管理界面

（3）切换到Service Hooks页面，如图16-14所示。Service Hooks页面提供了集成接口，让TFS可以在新代码提交或构建完成后通知第三方服务执行下一步任务。注意，这些服务并不局限于持续集成工具，只要其遵守TFS集成接口协议均可。单击Create a new subscription按钮创建一个新的集成任务。

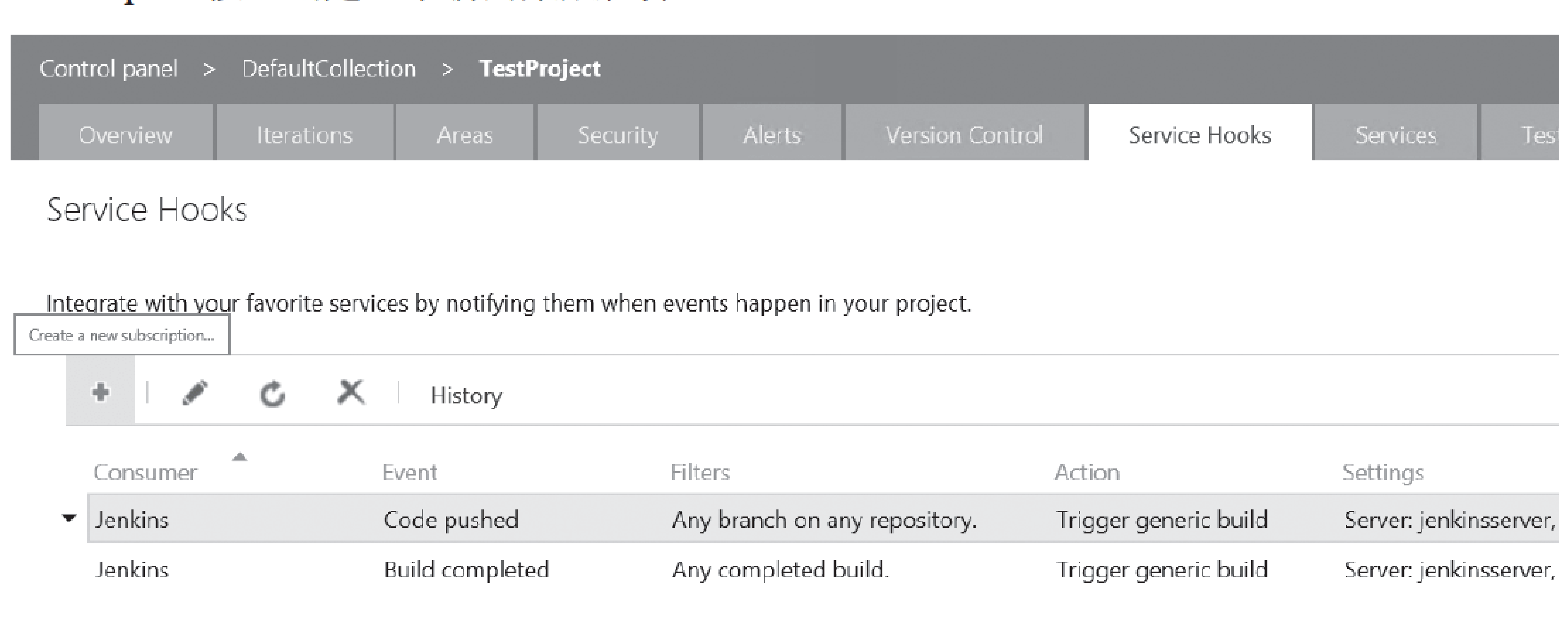


图16-14　Service Hooks页面

（4）如图16-15所示，在Service页面选择Jenkins选项，表示将与Jenkins进行集成。

（5）如图16-16所示，在Trigger页面选择触发类型为Code pushed，表示任何一次代码变更都会通知到Jenkins。另外，也可以选择Build completed表示一个版本构建完成。

（6）如图16-17所示，在Action页面内输入Jenkins服务的地址，用于访问Jenkins的用户名、密码以及对应的构建项名称。

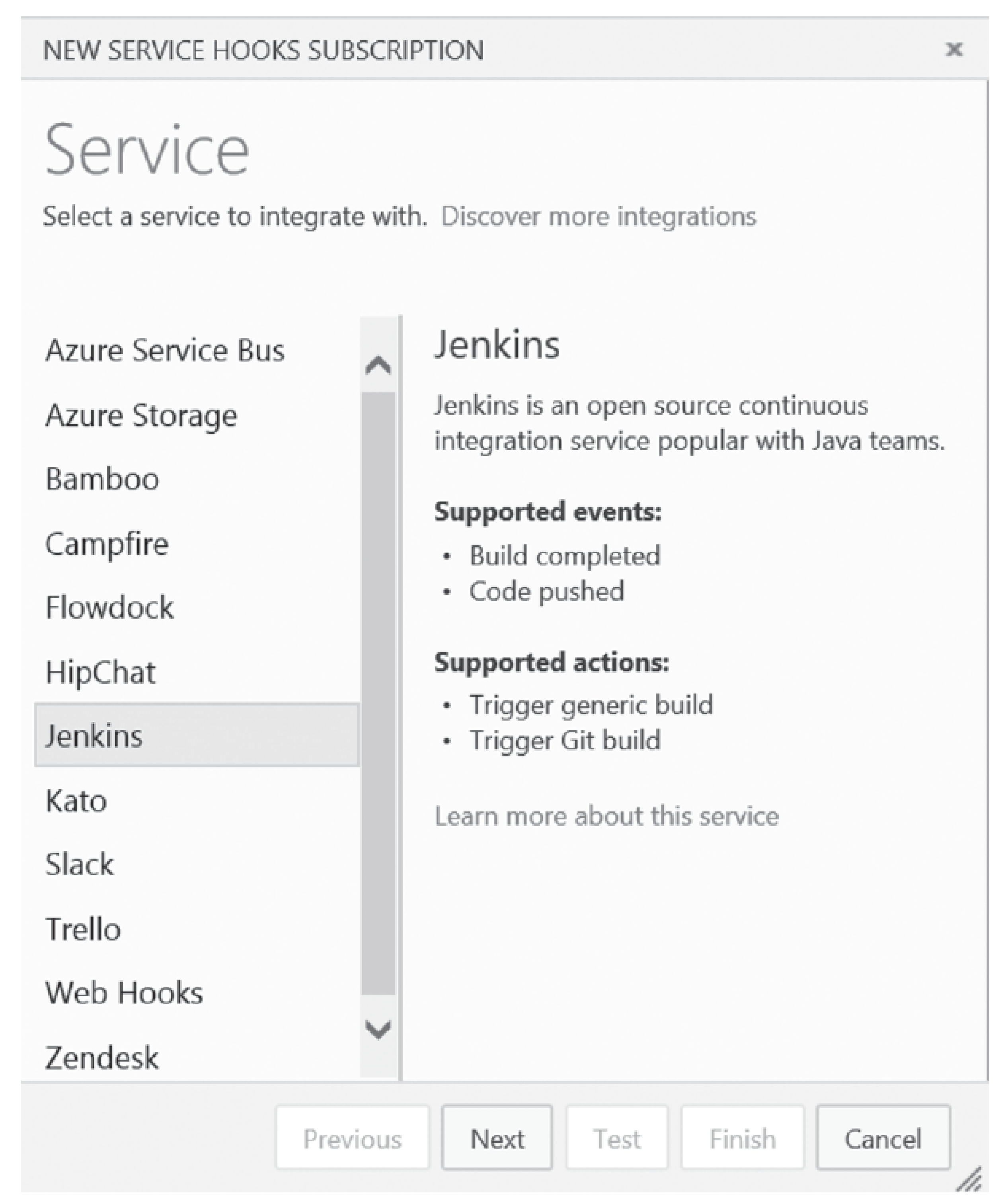


图16-15　选择第三方服务

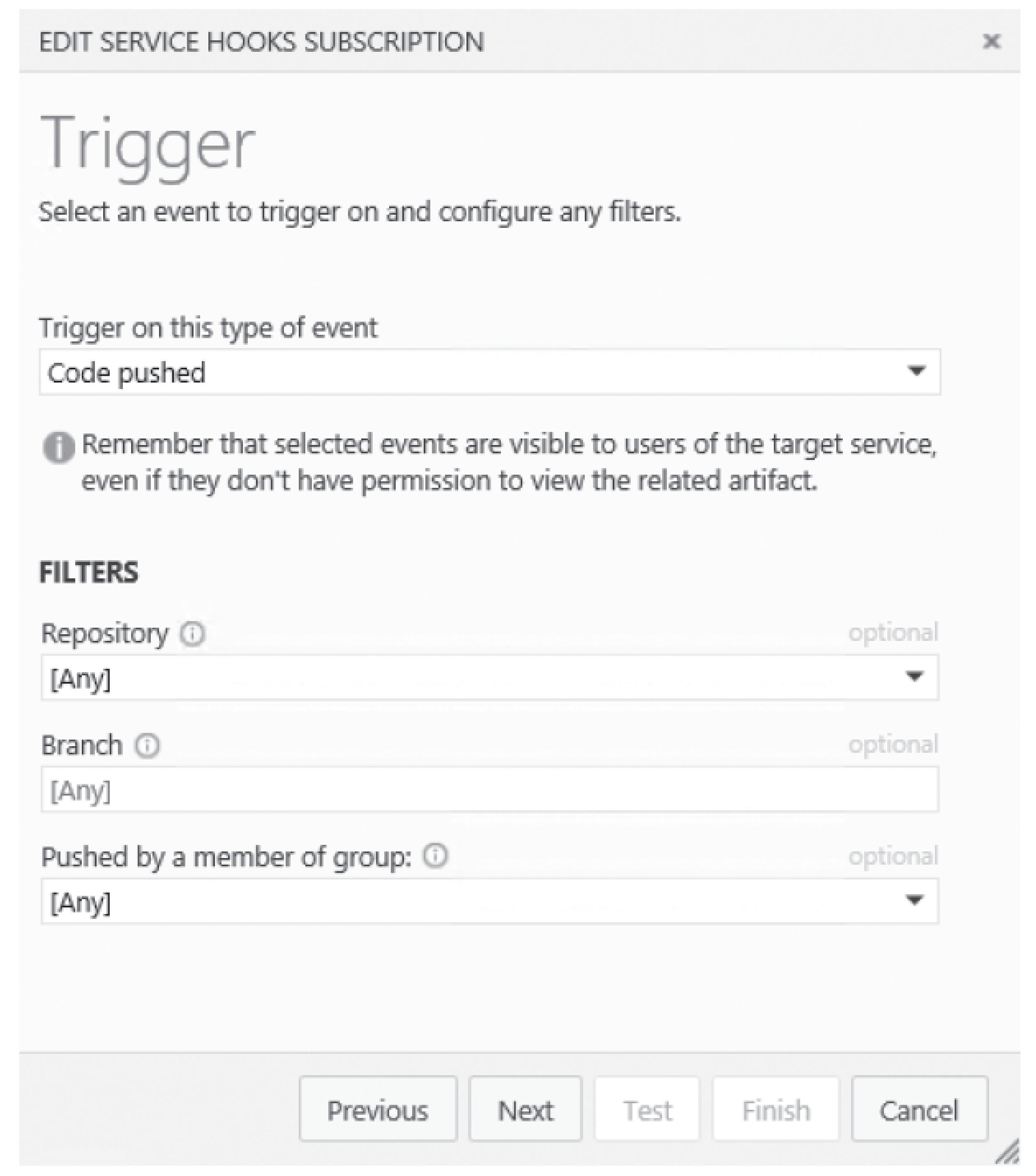


图16-16　选择Trigger类型

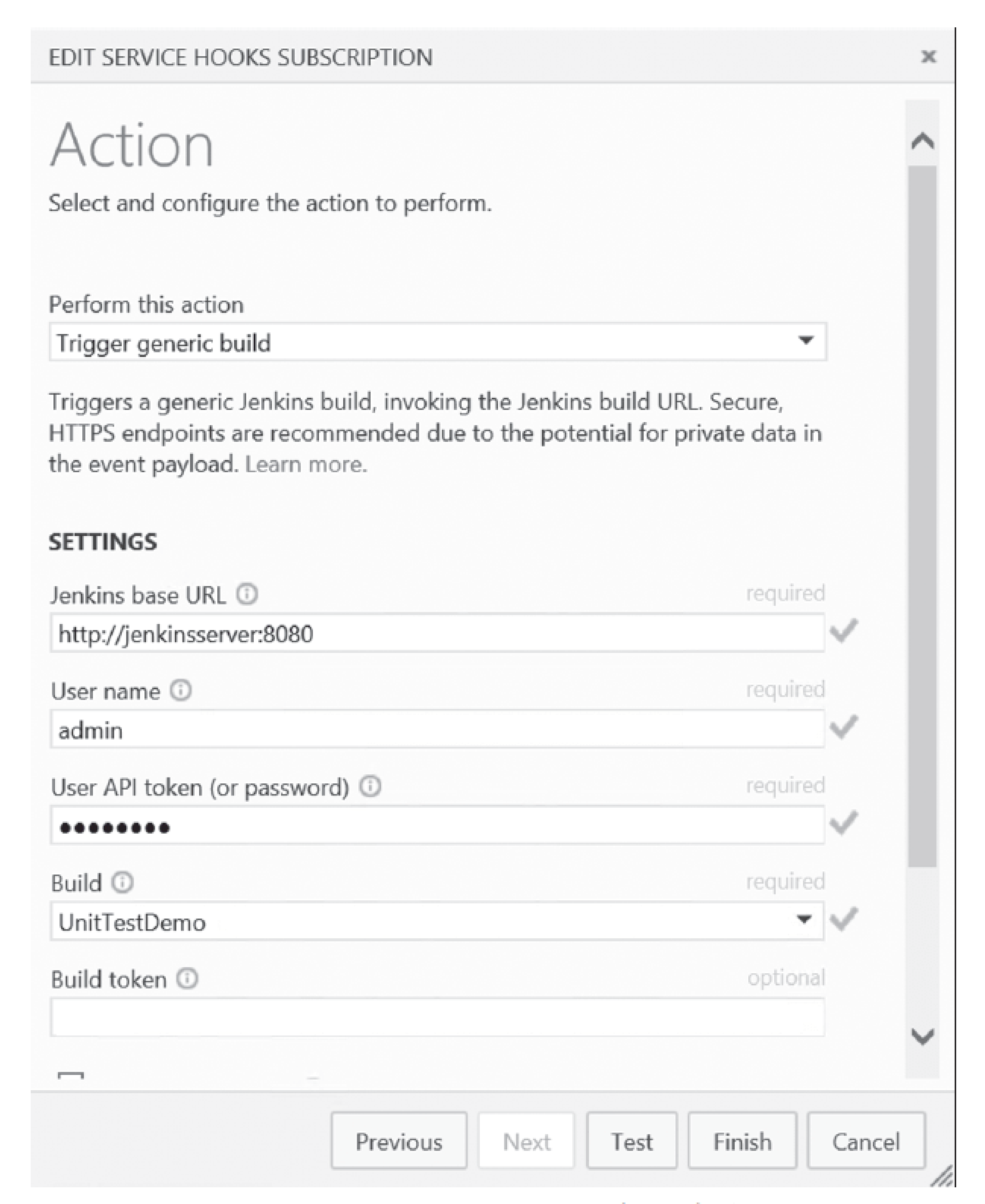


图16-17 配置Jenkins访问地址

（7）建议读者在每次创建或修改通知服务时，先在Action页面单击Test按钮检验配置是否正确。如图16-18所示，如果有403错误发生，请在Jenkins的Configure Global Security控制页面内取消勾选Prevent Cross Site Request Forgery exploits选项并再次测试。

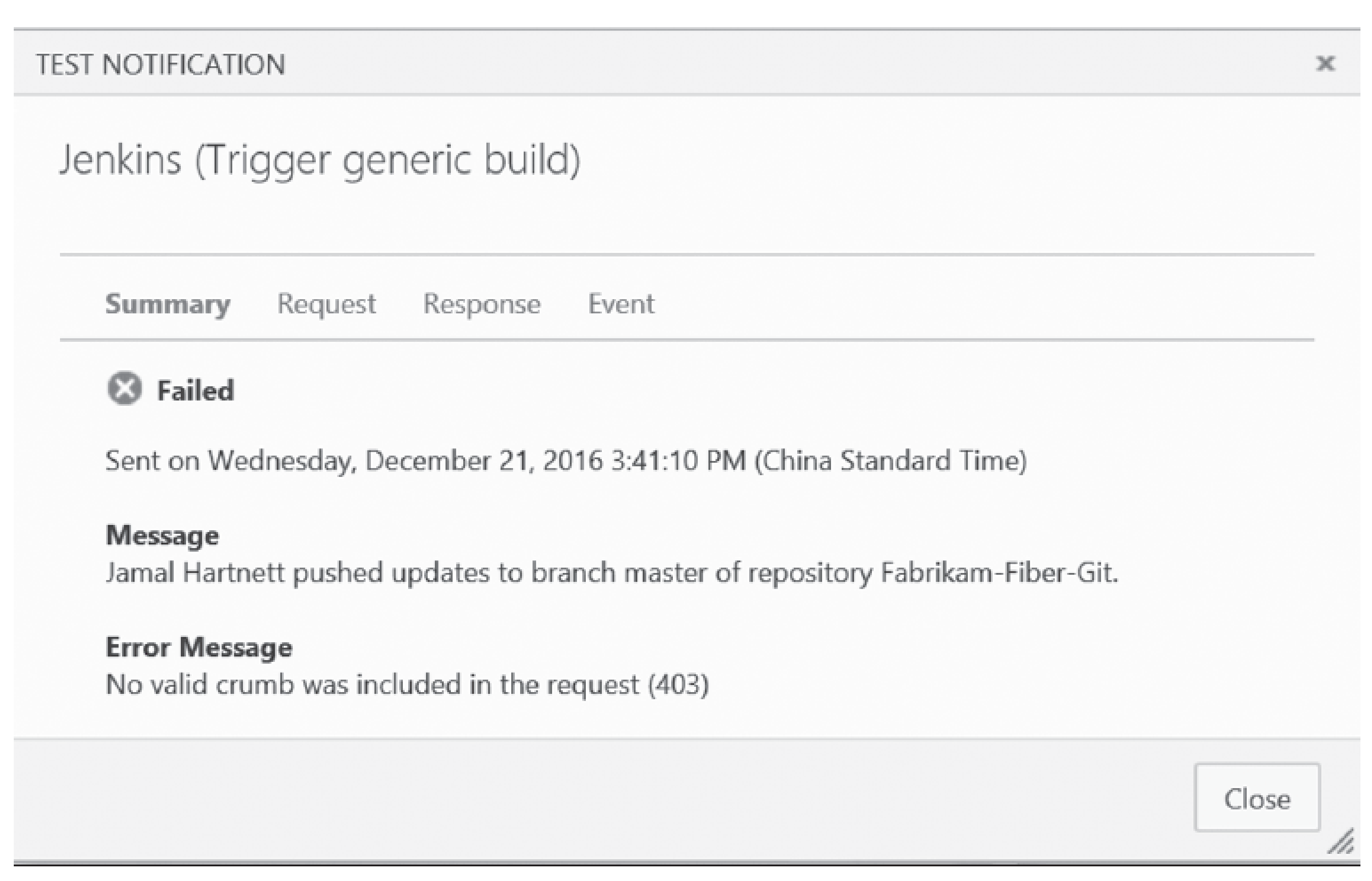


图16-18 出现403测试失败信息

（8）如图16-19所示，测试成功表示从TFS到Jenkins构建项的通知工作正常。

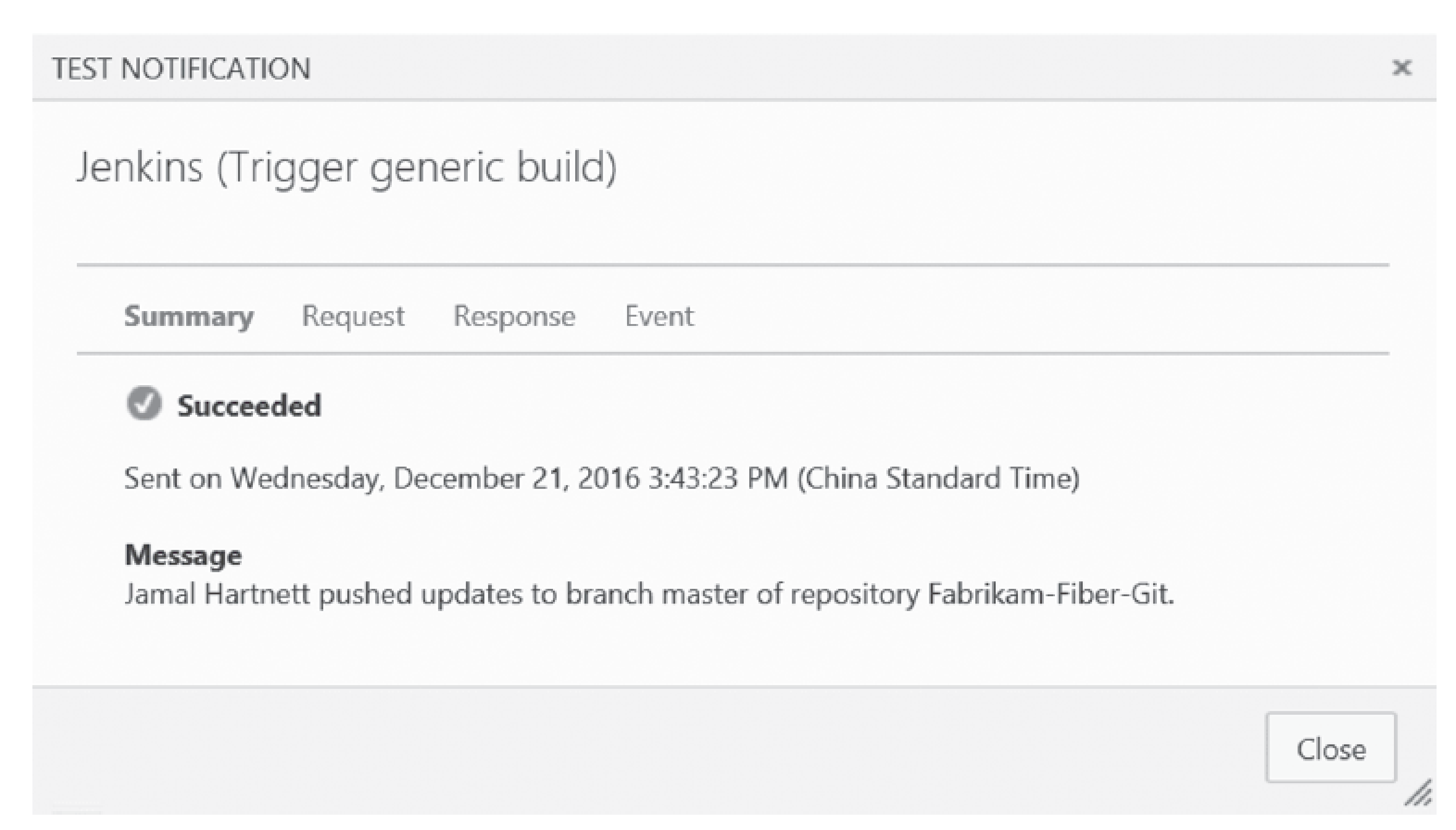


图16-19 成功连通

16.3.5 集成单元测试

集成单元测试的步骤如下：

（1）回到Jenkins服务器，在创建好的构建项UnitTestDemo中，找到Source Code Management页面，设置TFS代码仓库的访问链接和访问账号，如图16-20所示。

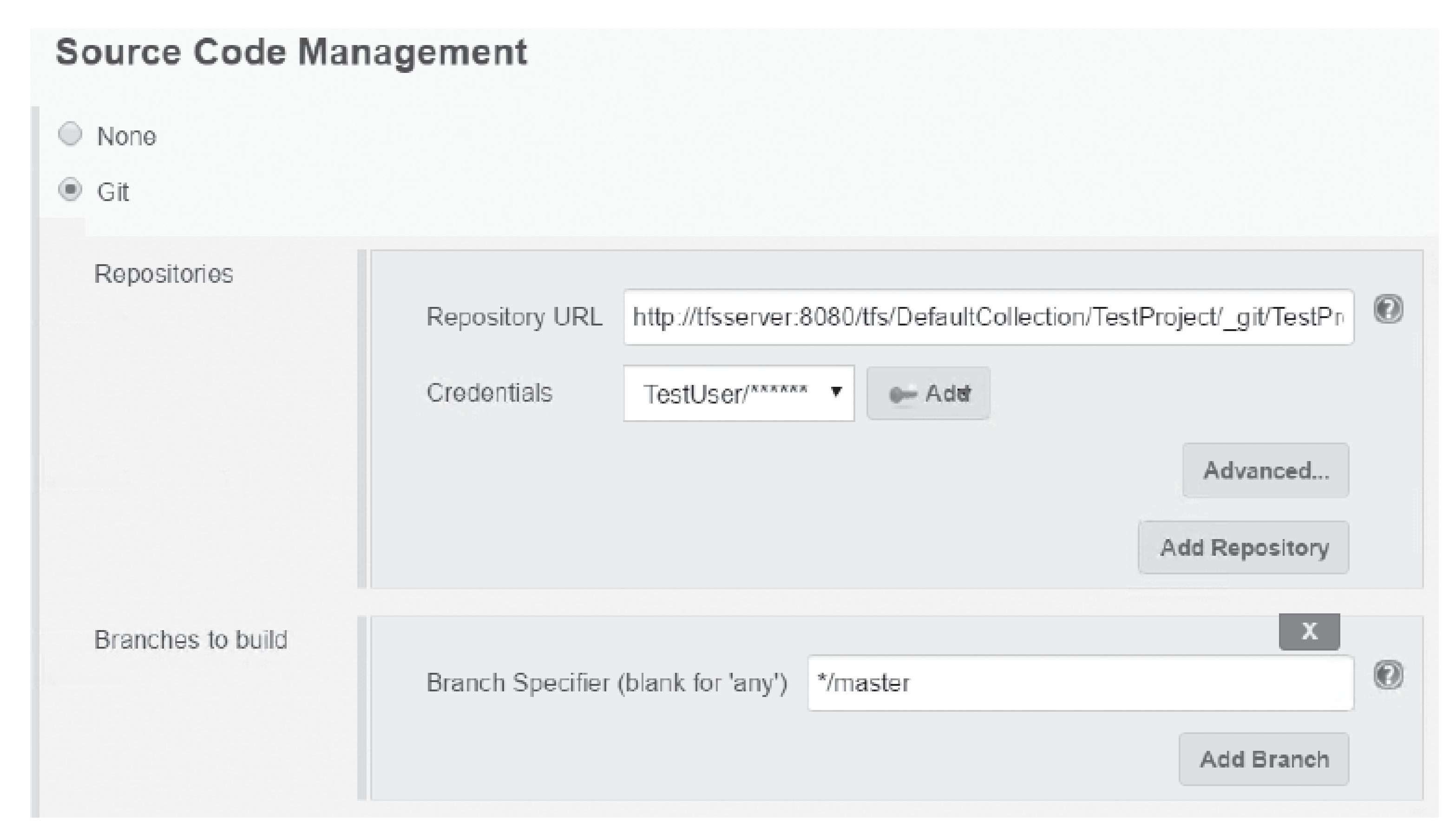


图16-20 配置源代码管理

（2）Jenkins的构建项可以基于不同的事件实现触发，例如在其他项目构建完成后触发、周期性触发等。作为单元测试，用户希望在每次有新代码提交到TFS后都进行构建。如图16-21所示，选择Build when a change is pushed to TFS/Team Services触发类型。

（3）完整的构建任务是由多个Build任务以定义好的顺序构成的。单击Add build step按钮可以添加多个Build任务。图16-22所示为按先后顺序调用npm install和npm run unit-test命令进行单元测试。

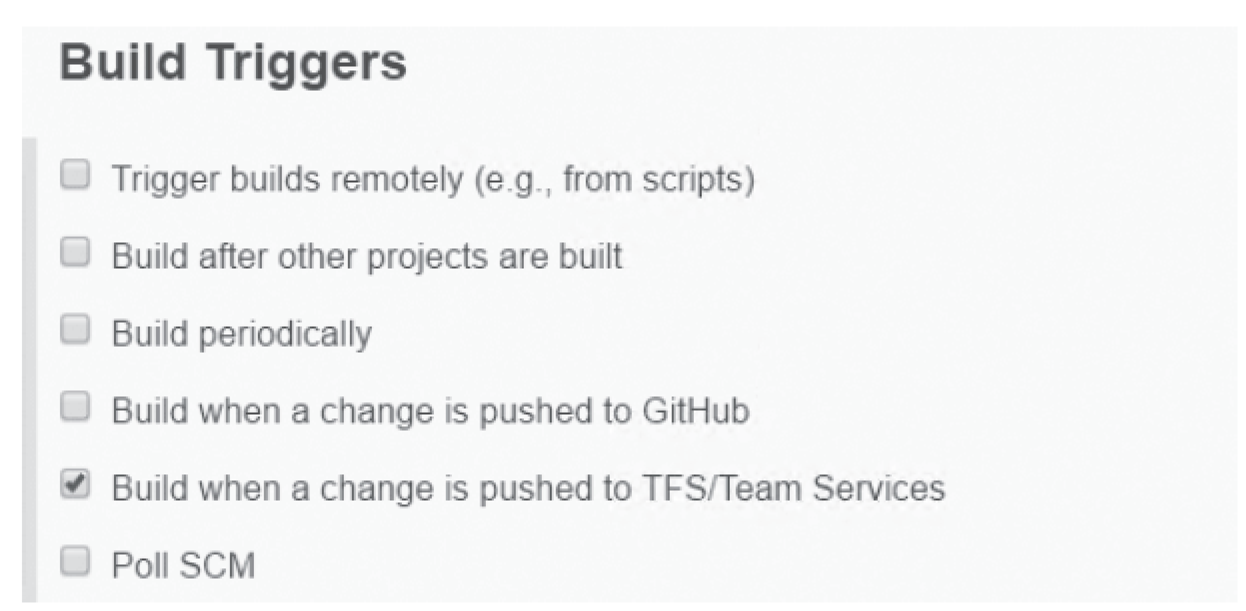


图16-21 选择触发类型

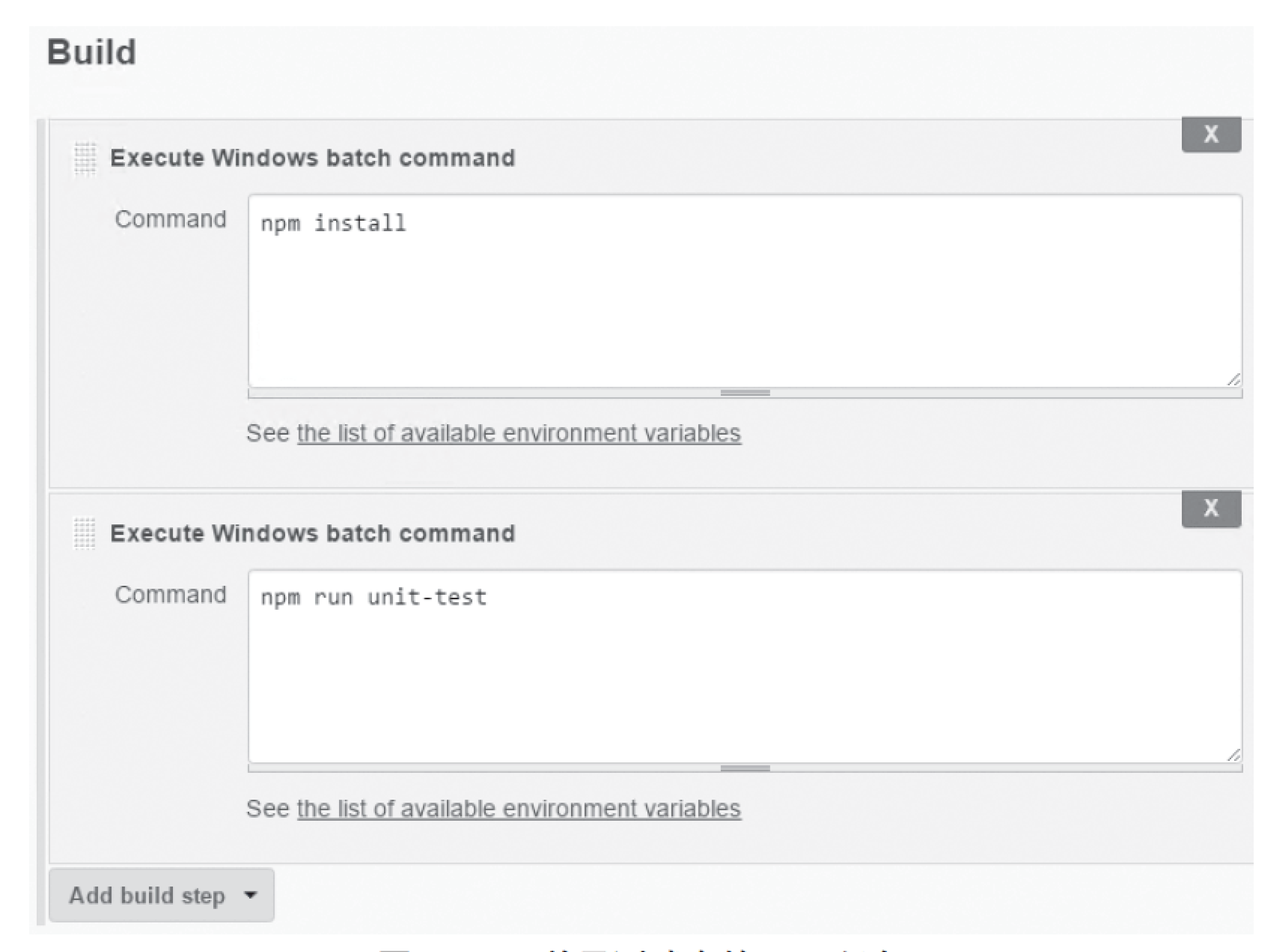


图16-22 单元测试中的Build任务



充分利用gulp和npm脚本进行构建，可以简化Jenkins构建项的配置。例如在npm脚本中将npm install配置为preunit-test任务，可以在Jenkins里省略对npm install的调用。

（4）完成并保存设置后，如果有任何新的代码变更被提交到TFS，都会触发Jenkins构建项。如图16-23所示，在Build History列表里可以看到已经完成和正在进行的构建。

（5）测试支持多种报告格式，其中JUnit报告被广泛用于持续集成中，用于查看测试结果。为了在Jenkins中使用JUnit，首先需要安装Junit Plugin插件（请参考16.3.3节中Team Foundation Server Plug-in的安装过程）。安装完成后，如图16-24所示，即可在构建项内通过单击Add post-build action按钮创建Publish JUnit test result report任务，并根据Karma中的配置项，指定对应JUnit报告的保存路径与文件名。完成以上配置后，Jenkins将能够识别测试生成的JUnit报告，并将其纳入到构建结果中。

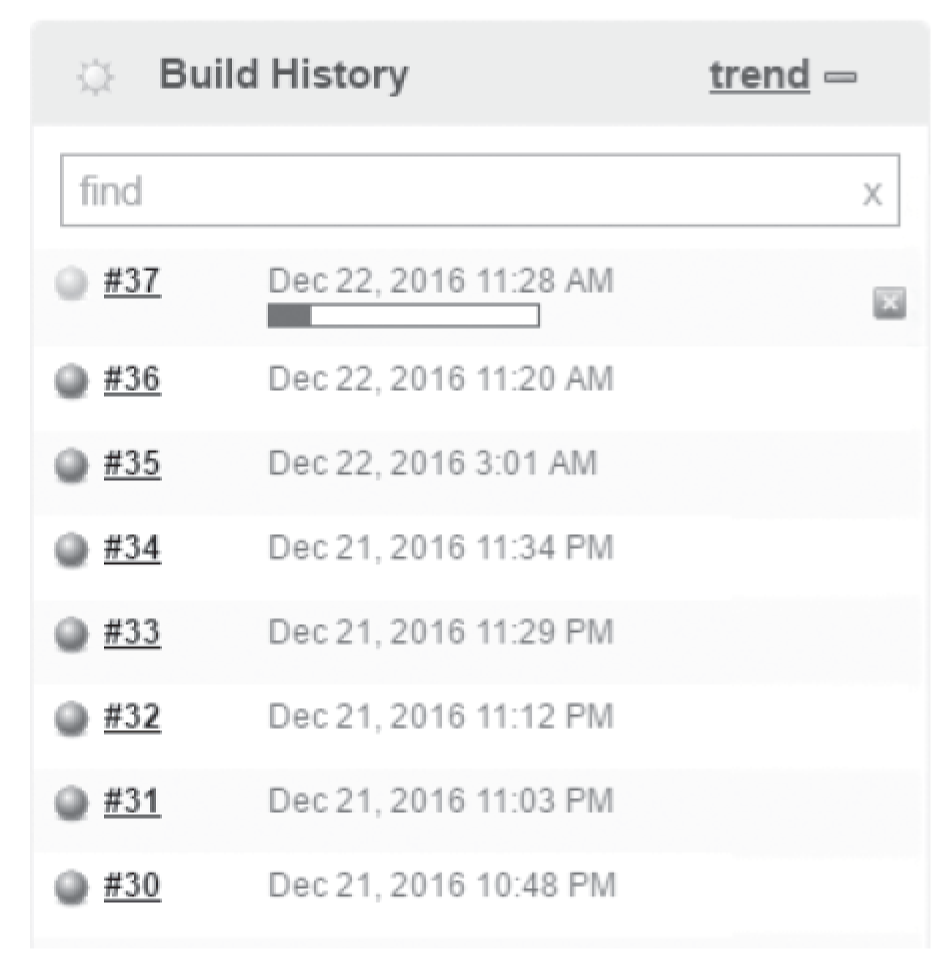


图16-23　构建历史列表

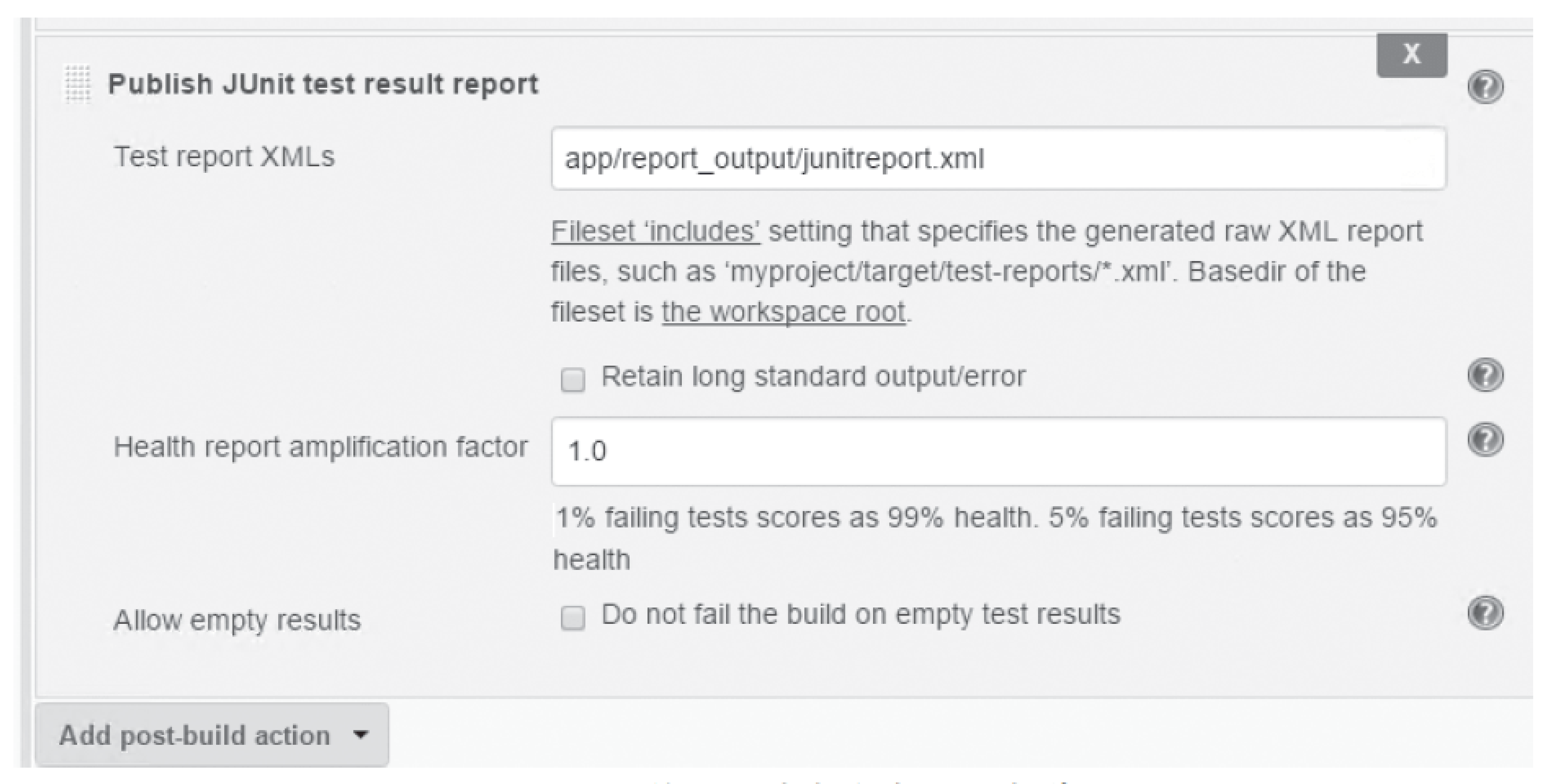


图16-24　单元测试中集成JUnit报告

（6）如图16-25所示，单元测试内的每个测试用例都被成功验证。测试用例的任何错误都会导致构建失败。

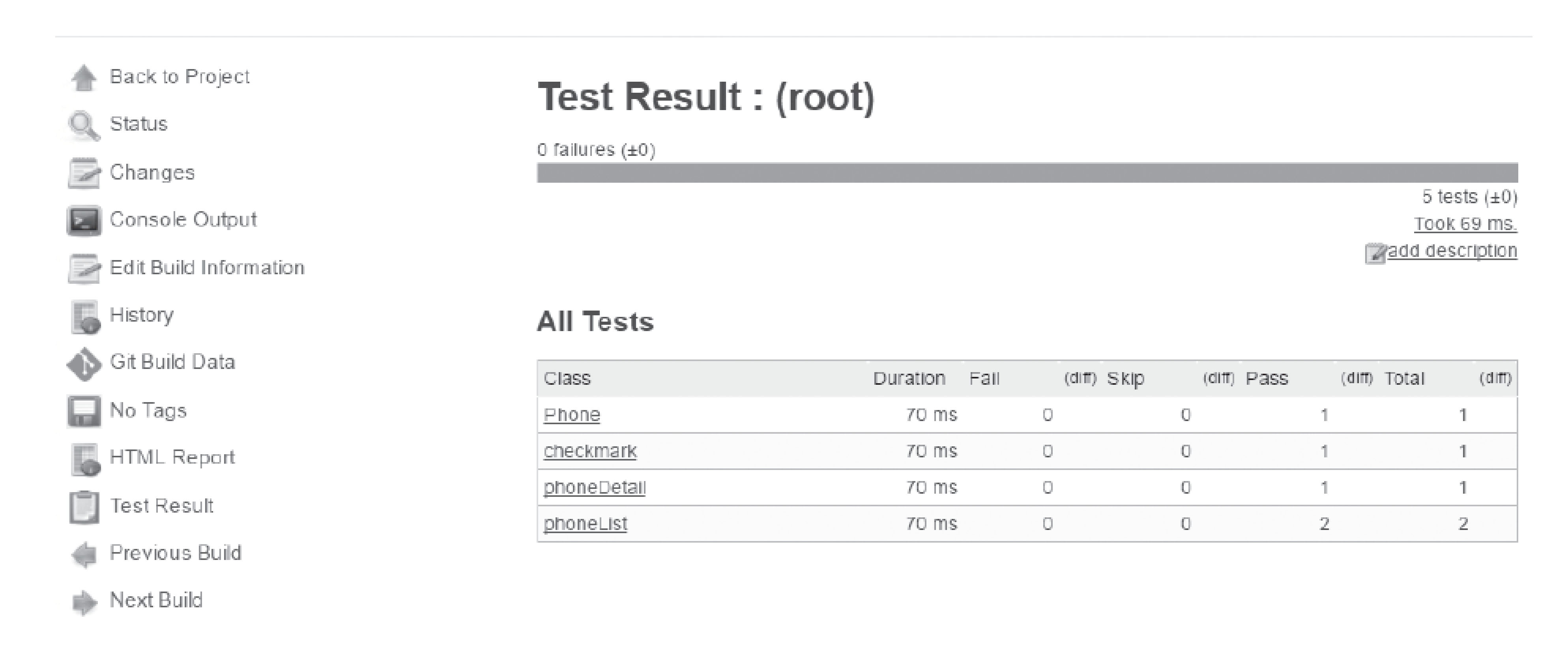


图16-25　单元测试结果

（7）除了JUnit报告，在安装了插件HTML Publisher plugin后（请参考16.3.3节中的安装过程），Jenkins还可以支持HTML报告，如图16-26所示。

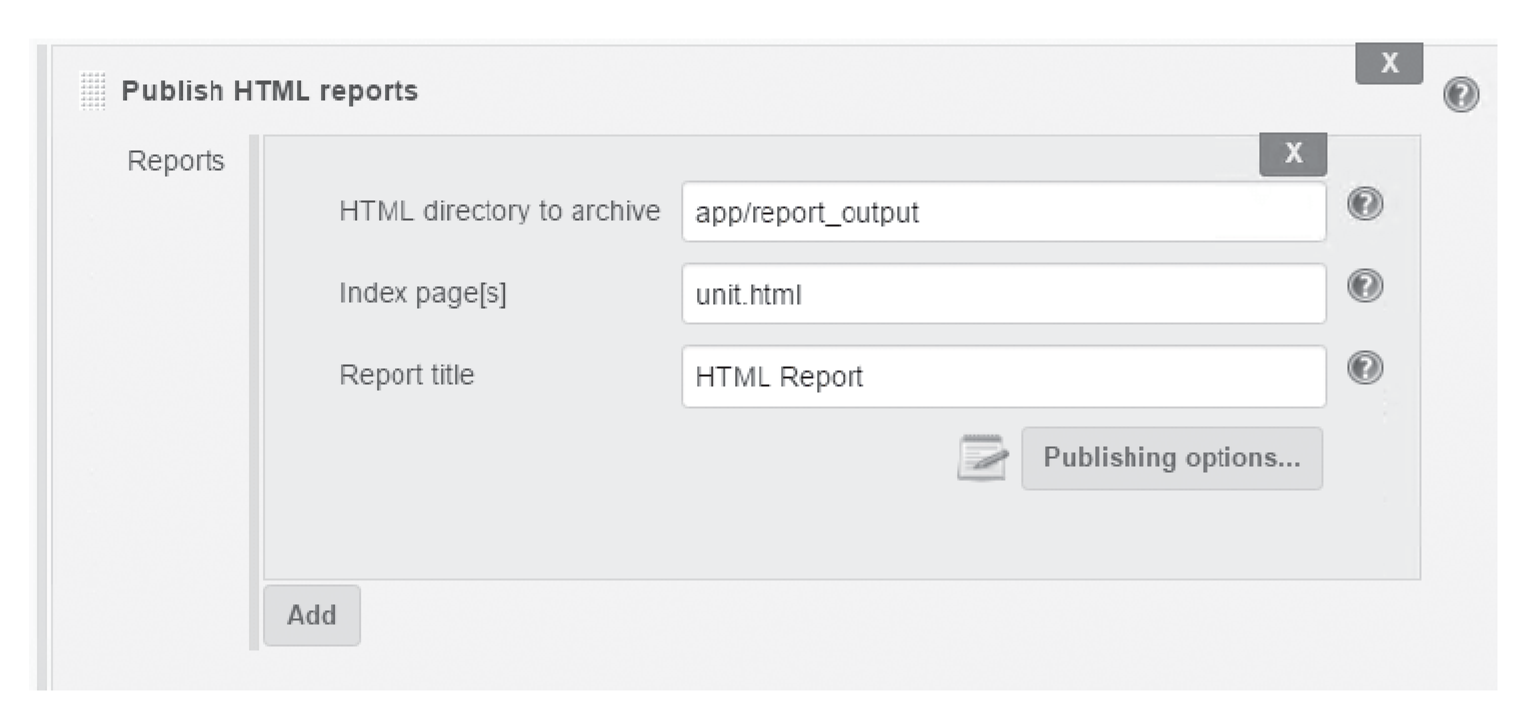


图16-26　单元测试中集成HTML报告

（8）在Jenkins内查看单元测试生成的HTML报告，如图16-27所示。

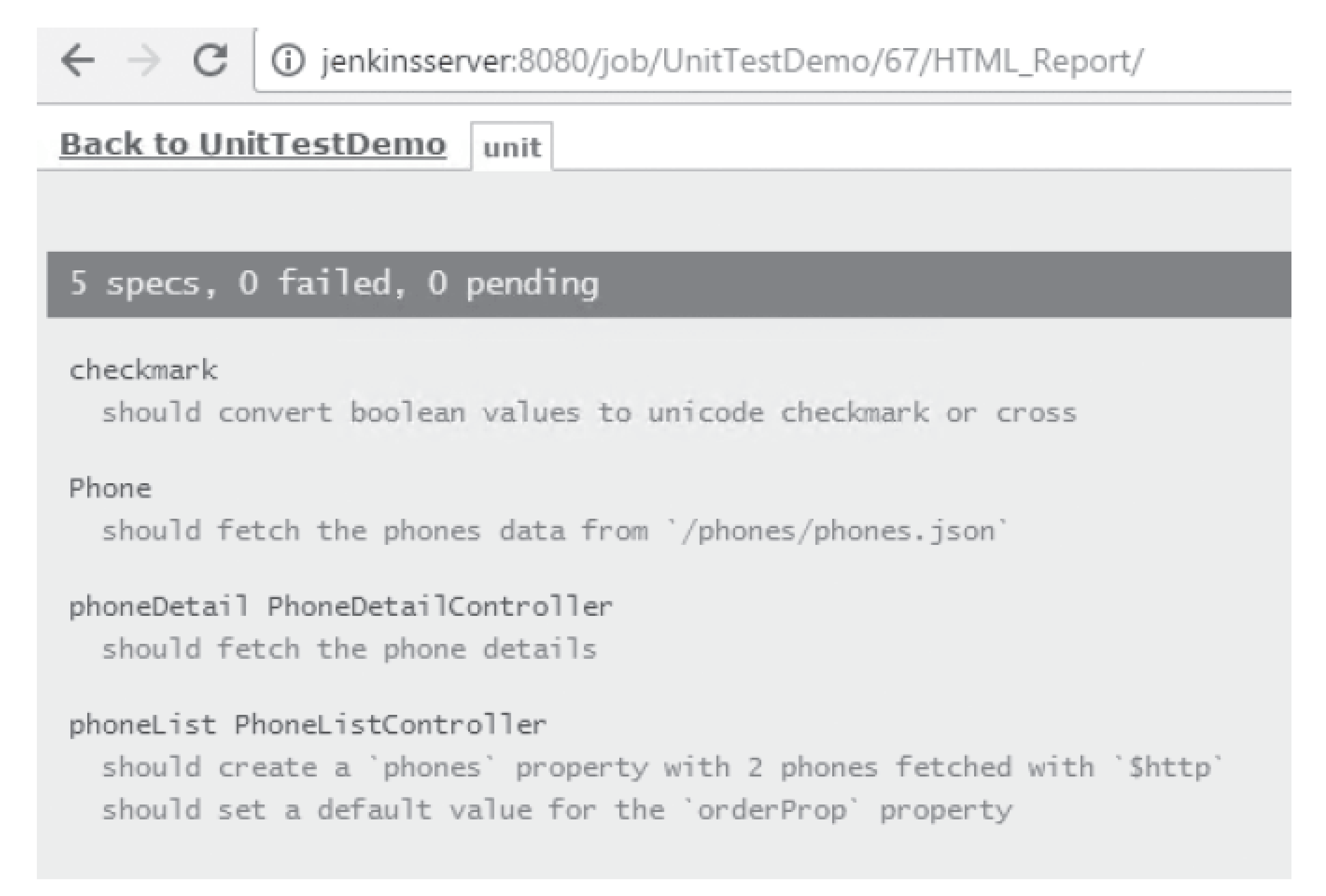


图16-27　显示HTML报告

基于最近若干次的构建结果，Jenkins会以不同的天气状态形象地表达构建的稳定指数[①]，如表16-1所示。这个功能使项目经理可以一目了然地了解当前的构建状态，并对下一步进展作出预测。

表16-1　构建稳定状态

状　态	描　述
	最近没有构建失败
	20~40%的构建失败

① Jenkins. Dashboard View[OL]. [2016]. https://wiki.jenkins-ci.org/display/JENKINS/Dashboard+View.

（续表）

状　态	描　述
	40~60%的构建失败
	60~80%的构建失败
	所有构建均失败

16.3.6 集成自动化测试

自动化测试与单元测试的主要区别是前者需要对软件进行部署后才能进行，而且测试用例本身执行速度比较慢。在这种情况下，如果每次更新代码都触发一次自动化测试则代价太高，一方面从构建服务器的性能考虑不太现实，同时也没有必要。

很多快速迭代的项目采取的方法是每日构建，即一般安排在晚上10点到12点之间出一个新的版本。之所以选择这个时间段是因为这个时候大多数员工已经下班，代码提交量偏少，是一个比较好的执行构建的窗口。当然，对于某些全球性项目，如采用中国、欧洲、美洲等不同时区接力式开发模式的项目，项目组可以根据自己的实际情况灵活选择适合的时间窗口。

进行每日构建既可以在TFS内完成，然后通过Build completed事件触发Jenkins的构建项，也可以直接在Jenkins内基于预定义的构建时间进行主动式触发。在创建好自动化测试构建项时，与单元测试类似，需要先设置好Git访问链接等选项。

为了实施每日构建，需要在Jenkins的Build Triggers页面内选择Build periodically选项并指定构建时间，格式与cron表达式类似。如果读者经常使用Linux，可能对Linux里的cron计划任务非常熟悉，它会根据命令和执行时间来按时执行调度任务。

Jenkins的时间计划由5个参数构成，中间以空格隔开，按顺序依次为：

- MINUTE：代表分钟，取值范围是0～59。
- HOUR：代表小时，取值范围是0～23。
- DOM：代表某天在当月内的数值，取值范围是1～31。
- MONTH：代表月，取值范围是1～12。
- DOW：代表某天在一个星期内的数值，取值范围是0～7，其中0和7都代表星期天。

如果需要指定多个值，可以采用如下操作数（优先级从上到下）：

- *：适配所有有效的值，若不指定某一项，则以*占位。
- M-N：适配值域范围，例如7-10代表7-/8/9/10均满足。
- M-N/X或*/X：以X作为间隔。
- A，B，C：枚举多个值。

另外，为了避免多个任务在同一时刻同时触发构建，可以指定在某个时间段内进行触发，需要配合使用H字符。添加H字符后，Jenkins会在指定时间段内随机选择一个时间点作为起始时刻，然后加上设定的时间间隔，计算得到后续的时间点。到下一个周期后，Jenkins又会重新随机选择一个时间点作为起始时刻，依次类推。

为了便于理解，以下举例说明。

- 0 * * * *：代表每个小时的第0分钟。
- H/15 * * * *：代表每隔15分钟，并且开始时间不确定。例如这个小时可能分别是08、23、38和53，而下个小时可能是02、17、32和47。
- H(0-29)/10 * * * *：代表前半小时内每隔10分钟，并且开始时间不确定。例如这个小时可能分别是04、14、24，而下一个小时可能是09、19和29。
- H 23 * * 1-5：工作日每晚23:00至23:59之间的某一时刻。

在本示例中，如图16-28所示，设置触发时间段为H(0-29) 23 * * 1-5，即每个工作日晚上23点前半小时内的某一时刻。

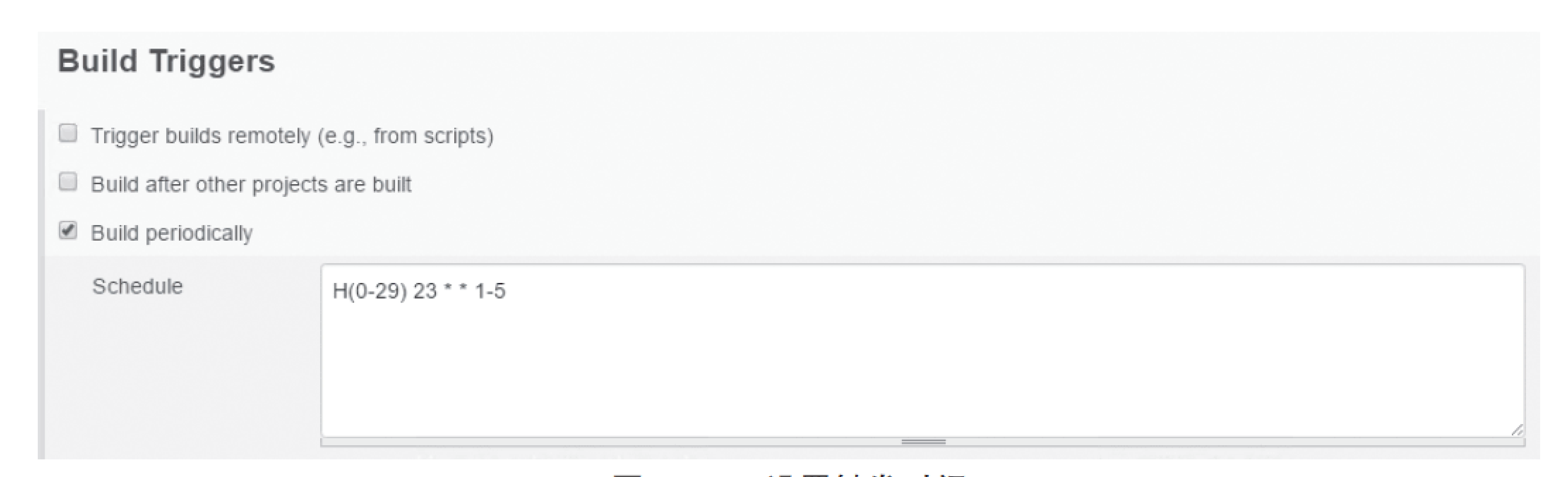


图16-28　设置触发时间

使用Protractor驱动自动化测试带来的一大好处是其技术栈与Karma完全一致，它们都是基于Node.js的，都可以通过gulp、Grunt进行构建，也都可以集成到npm脚本内。这样，在实施自动化测试集成的时候可以复用单元测试的知识，在部署完后使用类似的选项配置自动化测试。

如图16-29所示，与单元测试类似，可以直接通过npm脚本运行自动化测试。同样，如图16-30所示，也可以在自动化测试中集成HTML报告，只需要将对应路径配置正确即可。

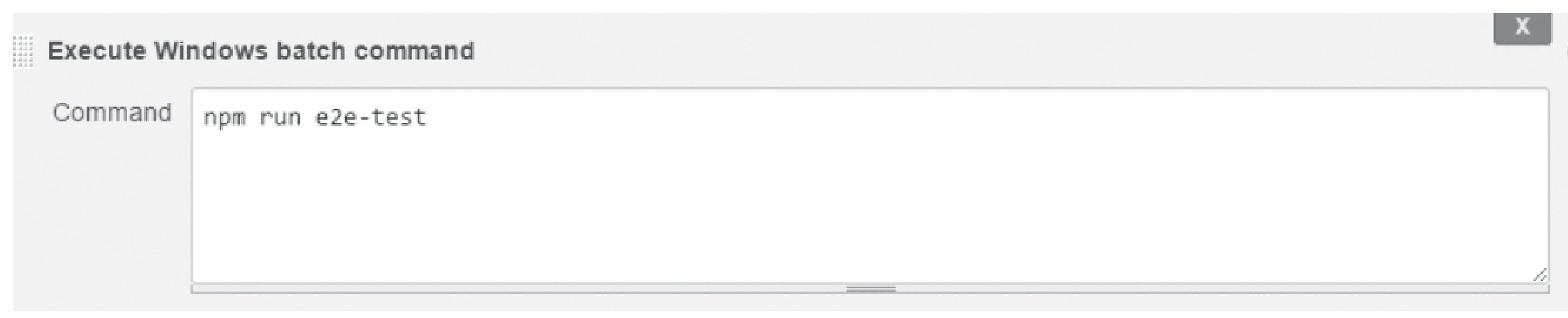


图16-29 向自动化测试中添加Build任务

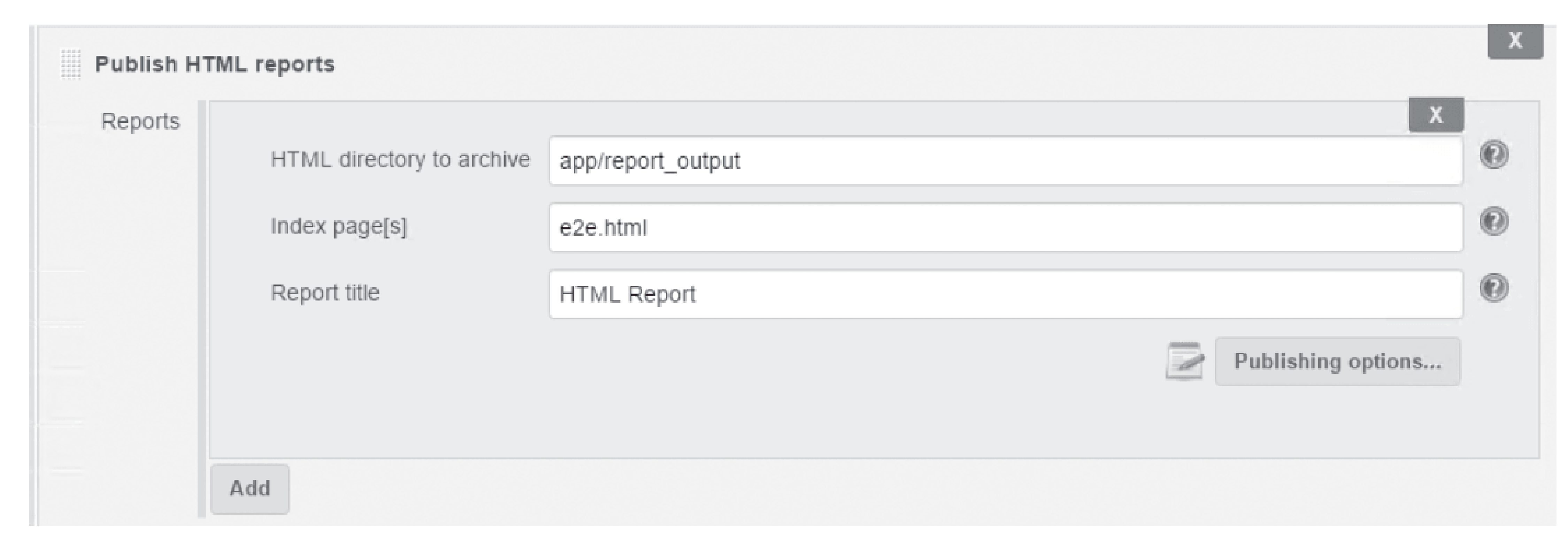


图16-30 在自动化测试中集成HTML报告

需要注意的一点是，自动化测试是一个消耗测试资源的耗时过程，不建议直接在Jenkins服务器上运行Selenium Server进行测试，而是采用Selenium Grid或Sauce Labs等云服务进行远程分布式测试。

16.3.7 邮件反馈

Jenkins可以将构建结果反馈给开发人员，为了支持邮件反馈，首先需要在Configure System里设置System Admin e-mail address，然后设置E-mail Notification选项，包括SMTP服务器、账号及密码等，如图16-31所示。

另外，Jenkins还支持增强版的Extended E-mail Notification，相比较而言它的功能更强大，可以定制邮件列表、文本内容，发送后执行脚本等。同时，它还支持多种触发条件，很适合对邮件定制要求较高的项目，如图16-32所示。

配置好邮件策略后，即可在对应的构建项中单击Add post-build action按钮添加Editable E-mail Notification任务完成邮件反馈的配置。

TFS和Jenkins在持续集成开发测试中使用广泛，功能强大，本书所介绍也只是其冰山一角。建议读者在实践中不断摸索，找到最适合自己项目所用技术的配置与实现办法。

E-mail Notification

SMTP server	smtp.contoso.com
Default user e-mail suffix	@contoso.com
☑ Use SMTP Authentication	
User Name	testadmin@contoso.com
Password	•••••••••••
Use SSL	☑
SMTP Port	465
Reply-To Address	testadmin@contoso.com
Charset	UTF-8
☐ Test configuration by sending test e-mail	

Save Apply

图16-31 配置邮件反馈

- ☐ Aborted
- ☐ Always
- ☐ Before Build
- ☐ Failure - 1st
- ☐ Failure - 2nd
- ☑ Failure - Any
- ☐ Failure - Still
- ☐ Failure - X
- ☐ Failure -> Unstable (Test Failures)
- ☐ Fixed
- ☐ Not Built
- ☐ Script - After Build
- ☐ Script - Before Build
- ☐ Status Changed
- ☐ Success
- ☐ Test Improvement
- ☐ Test Regression
- ☐ Unstable (Test Failures)
- ☐ Unstable (Test Failures) - 1st
- ☐ Unstable (Test Failures) - Still
- ☐ Unstable (Test Failures)/Failure -> Success

图16-32 邮件反馈触发类型

16.4 集成Visual Studio Team Services

Visual Studio Team Services（VSTS）[①]的前身即Visual Studio Online，是微软基于云计

① Microsoft. Visual Studio Team Services[OL]. [2016]. https://www.visualstudio.com/team-services/.

算，为广大开发人员推出的软件全生命周期管理的PaaS服务，是目前跨多种平台进行规划、构建和交付软件的最快捷的方式。

基于VSTS服务，用户数分钟内就能在微软云服务架构上启动VSTS并开始工作，无须再自己安装或配置单独的服务器。VSTS的操作界面与TFS非常相似，它将新的功能、缺陷和其他工作项捕获到待办事项中，对于运用Scrum、看板或自有灵活流程的团队而言，创建待办事项非常合适。另外，也可以使用自定义的任务板来跟踪团队进度，或者使用灵活的组合管理让更大的小组跟踪其所有团队的工作。



用户数不超过5人的团队可以免费使用VSTS。

VSTS支持Git客户端和Web浏览器界面，如图16-33所示。无论开发人员使用什么操作系统，哪种开发工具，如Visual Studio、Visual Studio Code、Eclipse、IntelliJH和Android Studio等，VSTS都可以完全支持。



图16-33　VSTS支持的开发工具

作为一个完整的生命周期管理服务，VSTS不仅支持版本和代码管理，也支持持续集成，包括测试、部署和发布等全部功能，但有些团队可能仍然希望继续使用自己已经熟悉的Jenkins做持续集成服务。本节后续将演示如何对VSTS和Jenkins进行集成。实际上，VSTS底层使用的是TFS解决方案，对于已经用过TFS的开发人员几乎可以直接上手，没有任何学习难度。

（1）使用VSTS前，首先需要有一个微软账号，可以到网址https://signup.live.com处申请。

（2）登录网址https://visualstudio.com即可创建VSTS服务，如图16-34所示。

（3）创建好的VSTS账户主界面如图16-35所示，与TFS的界面很接近。

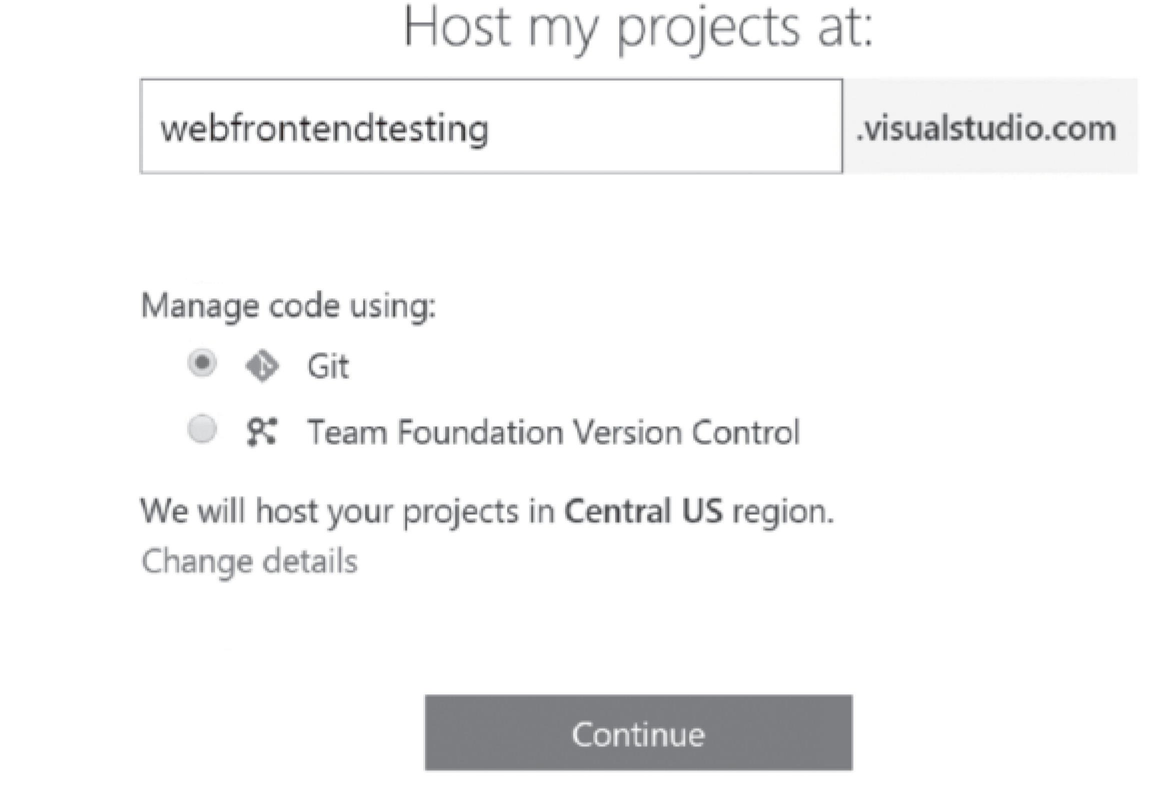


图16-34　创建VSTS账号

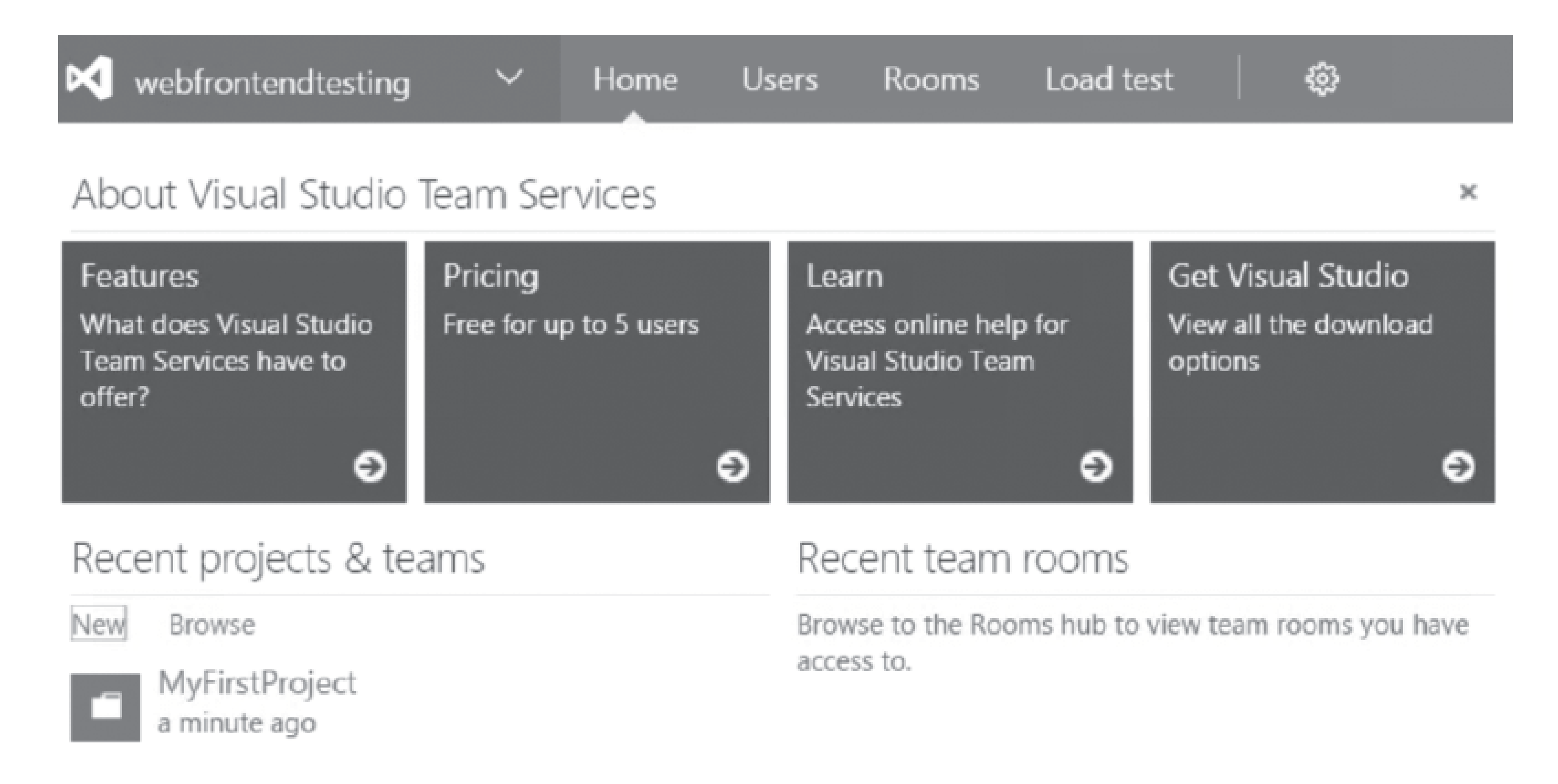


图16-35　VSTS账户主界面

（4）图16-36为具体项目的管理页面，提供了代码管理、持续集成和报表等功能入口，并且可以邀请新的项目成员加入。

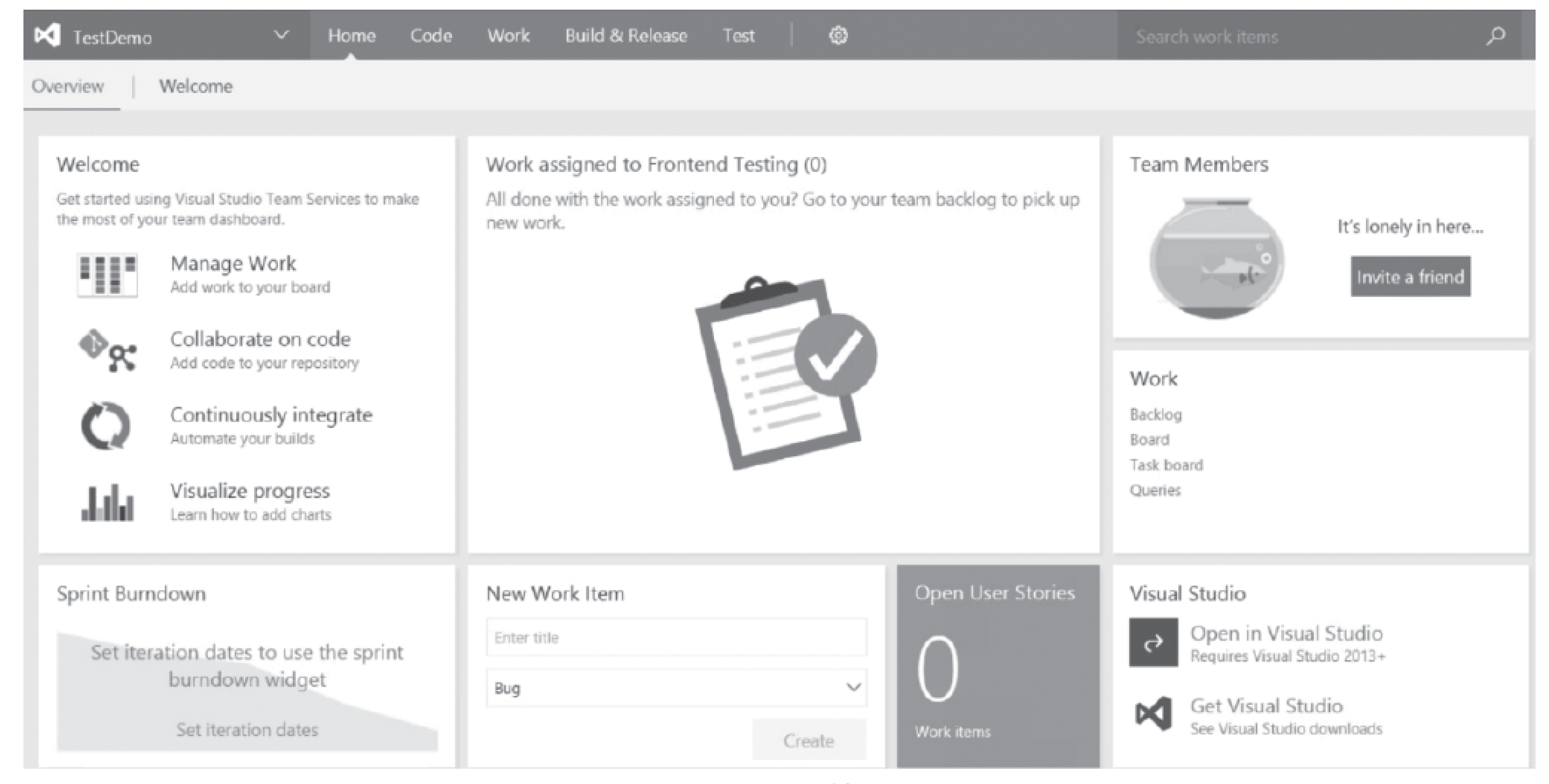


图16-36　项目管理页面

（5）出于安全考虑，VSTS不允许 Jenkins通过微软账号和密码连接VSTS，解决方案是创建一个替代账户。如图16-37所示，在账户管理界面右上角，单击用户图标，选择My profile选项。然后选择Security→Alternate authentication credentials选项进入相应的页面，选择Enable alternate authentication credentials选项，并创建secondary用户和密码。该用户将用来连接VSTS和Jenkins，如图16-38所示。

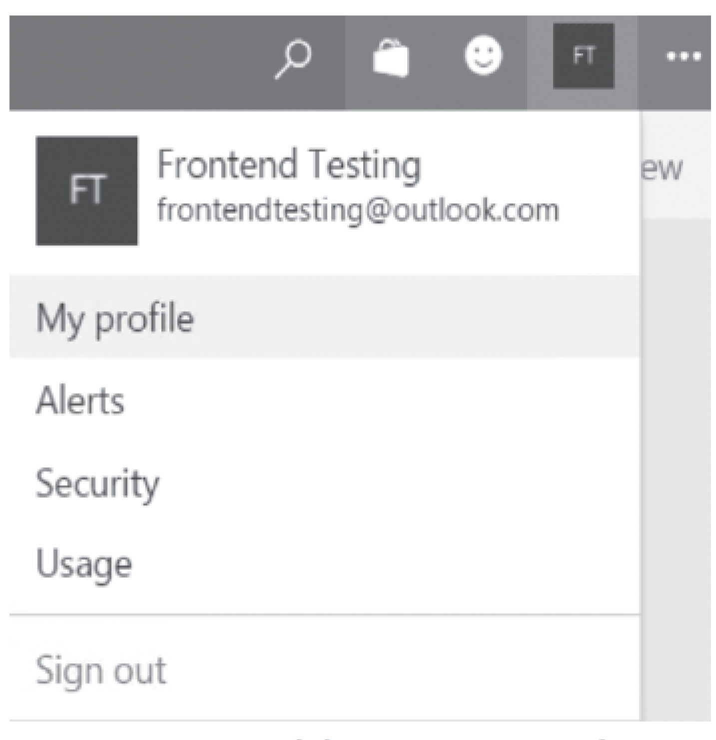


图16-37　选择My profile选项

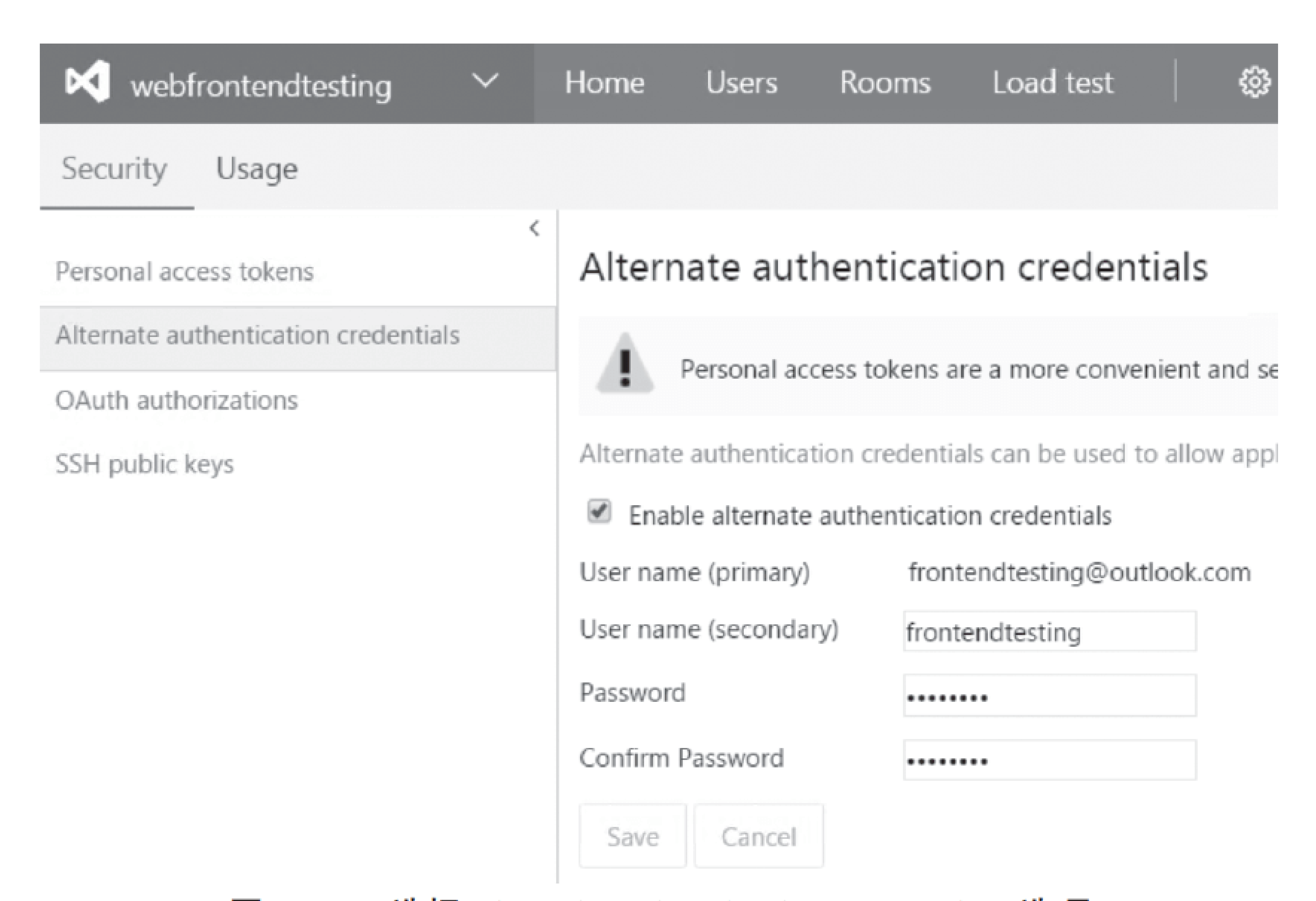


图16-38　选择Alternate authentication credentials选项

（6）回到Jenkins项目配置页面。VSTS的集成与TFS类似，也是通过Team Foundation Server Plug-in插件实现的。需要注意的是在配置Git代码仓库的时候，输入的不是微软账号，而是已经创建的secondary用户的账号，如图16-39和图16-40所示。

（7）回到VSTS的Service Hooks页面。为了让VSTS的Service Hooks在发生变更时能够通知到Jenkins，VSTS需要能够通过公网IP访问Jenkins服务器。在本示例中，作者把Jenkins部署到了微软公有云Azure上的虚拟机，从而可以被VSTS访问到。

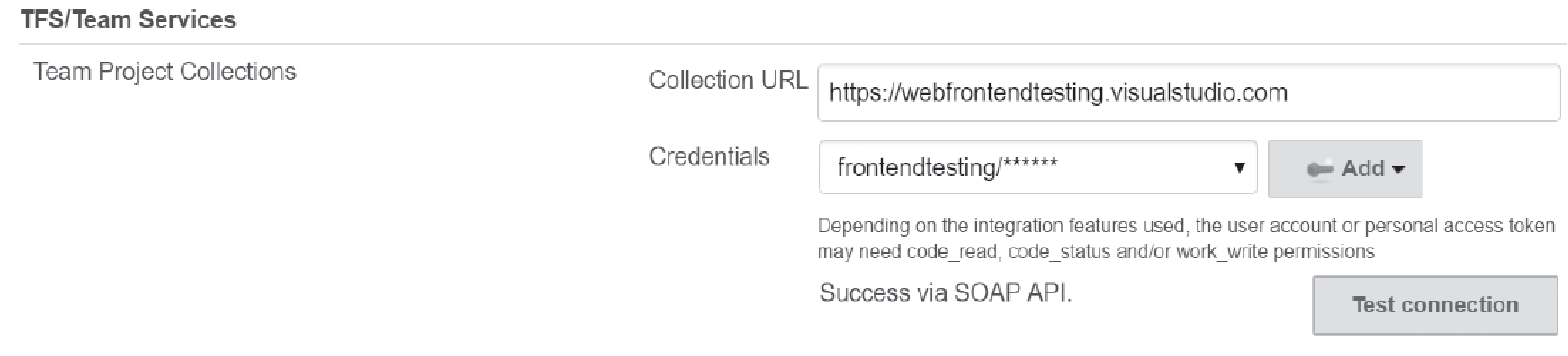


图16-39　配置VSTS Credentials

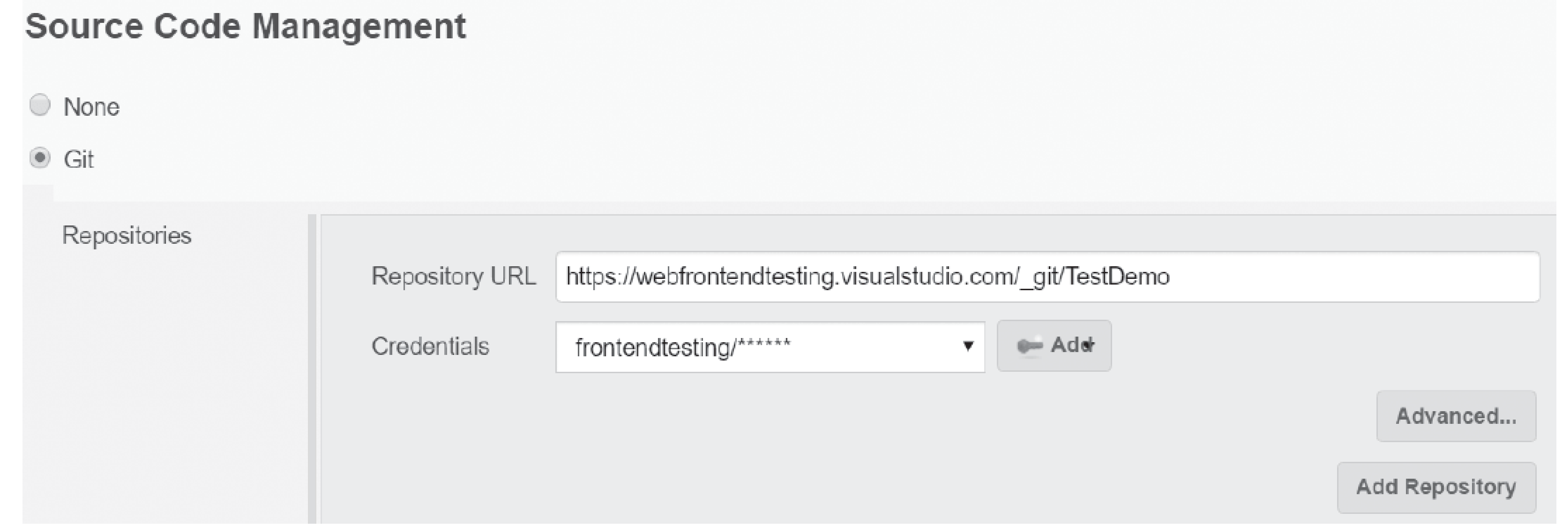


图16-40　配置源代码管理



如果Jenkins持续集成服务器被部署在公司内网，可以考虑使用Nginx或者Application Request Routing（ARR）工具通过反向代理将其发布到外网。

（8）在Service Hooks页面设置用户时，既可以用密码也可以用API Token，从最佳实践的角度作者更推荐后者。如图16-41所示，API Token可以在Jenkins用户设置中，通过单击Show API Token按钮获得。

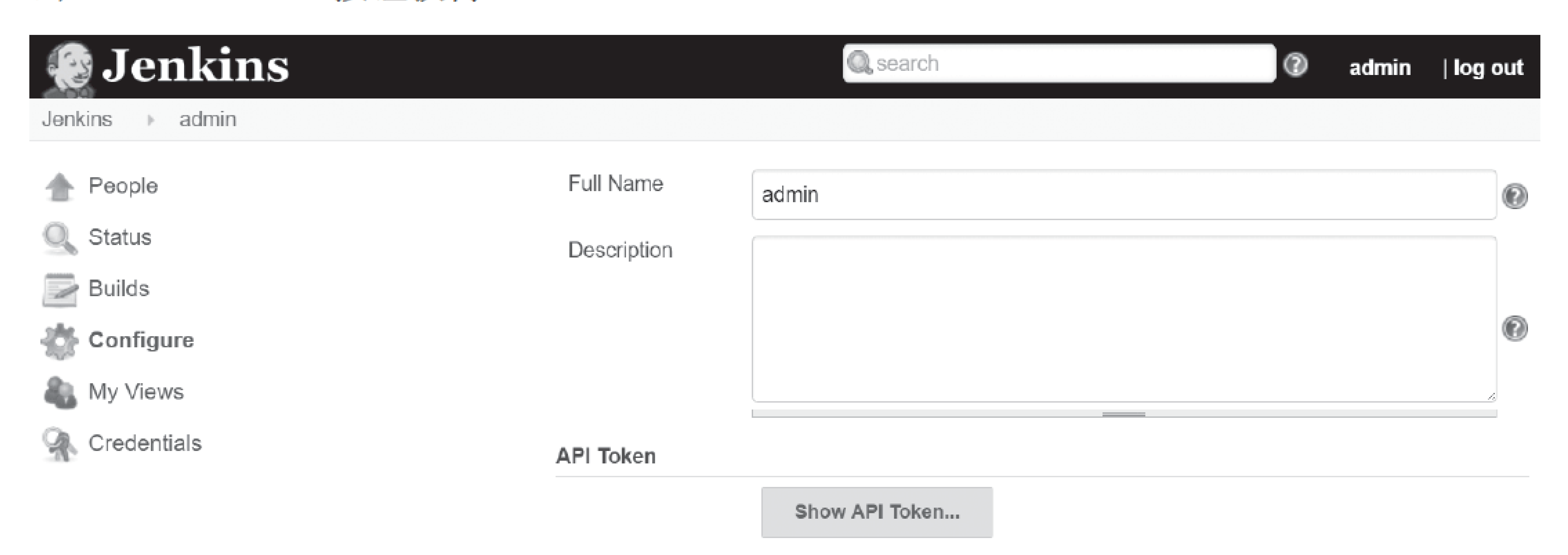


图16-41　获得API Token

（9）如图16-42所示，在Service Hooks页面，设置Jenkins base URL、用户名、密码、Build项目名称，Integration level选择Built-in Jenkins API选项。

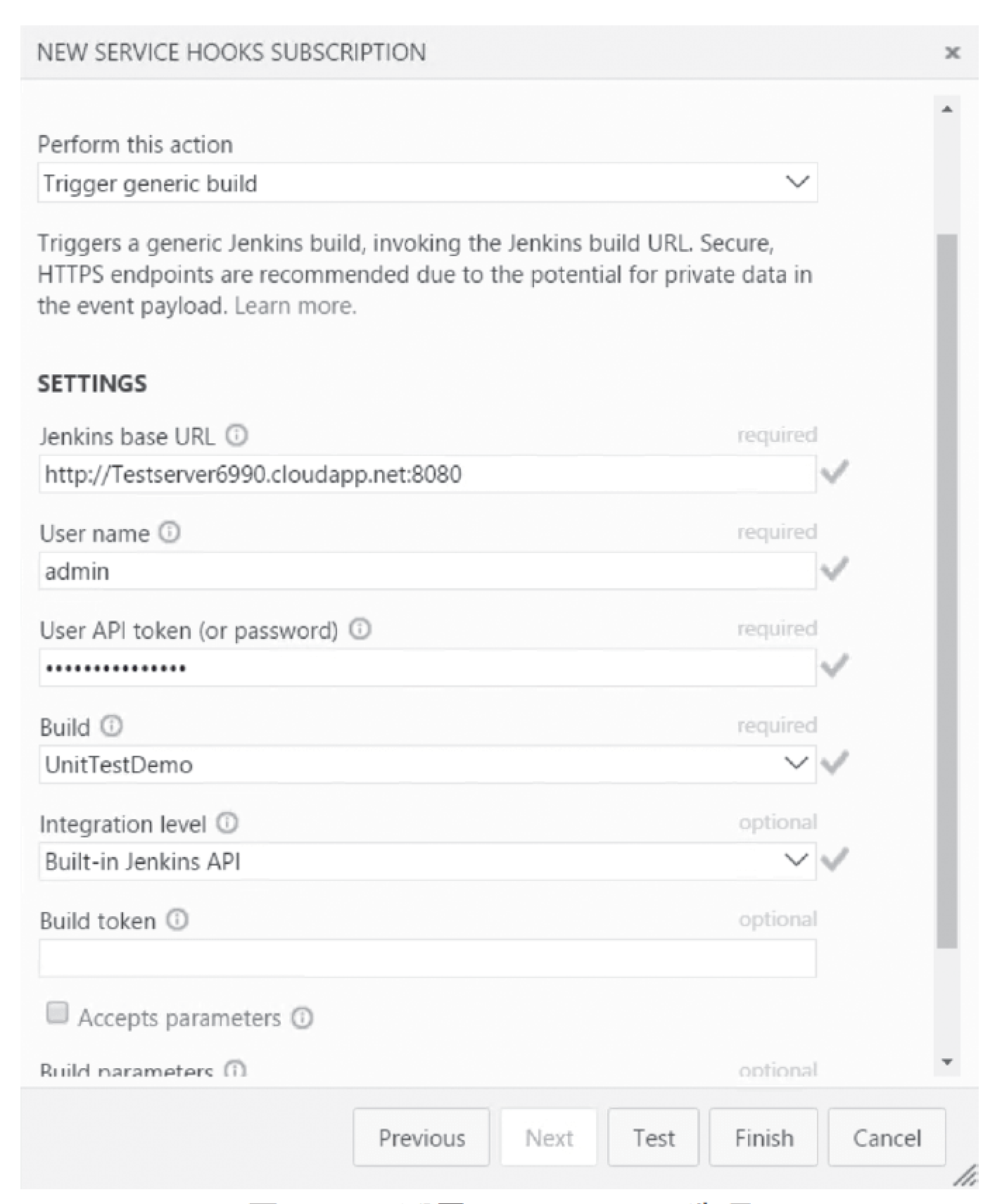


图16-42 配置Service Hooks选项

其余关于单元测试和自动化测试的集成步骤，与上一节TFS一致，这里不再赘述。

16.5 集成GitHub

GitHub是一个面向开源及私有软件项目的托管平台，因为只支持 Git 作为唯一的版本库管理格式，故名 GitHub。

GitHub于2008年4月10日正式上线，除了 Git 代码仓库托管以及Web 管理界面以外，还提供了订阅、讨论组、文本渲染、在线文件编辑器、协作图谱和代码片段分享（Gist）等功能。

GitHub还是一个开源协作社区，通过GitHub，既可以让别人参与你的开源项目，也允许你参与别人的开源项目。在GitHub出现以前，开源项目开源容易，但让各地的程序员参与进来却比较困难。因为要参与，就要提交代码，而给每个想提交代码的开发人员都开一

个账号并不方便，也没有统一的实现手段。所以，大多数情况下，即便开发人员想参与，也主要限于提交缺陷报告；即使能修改缺陷，也只能通过邮件发送代码，对大型项目很不方便。GitHub通过强大的fork和pull request功能，让世界各地的开发人员为项目贡献代码并使申请代码合并变得非常简单，这样广大开发人员便真正可以自由地参与到各种开源项目中，大大节省了沟通和协作成本。

通过不断进行项目的分支、合并和参与，开发人员在GitHub中的活动就如同交友一样，社会关系图的节点在不断地连线与发展，如今GitHub已经成为全球最大的社交编程与代码托管服务商。在作者编写本书的时候（2016年12月），GitHub上已经有1900万开发人员在开发、共享代码，项目数量超过了5000万个。难怪GitHub的首席执行官Chris Wanstrath曾经形象地将GitHub称为“程序员的维基百科全书”。

除了开源项目，GitHub也支持付费的私有库和企业版本（GitHub Enterprise）。其中企业版本的GitHub主要面向大型公司，可以将其私有库置于企业防火墙之后。

GitHub使用非常广泛，因此接下来本书将介绍把GitHub和Jenkins集成起来的方法。与TFS和VSTS类似，GitHub上有任何新的代码变更都可以通知到Jenkins，Jenkins也可以周期性地执行构建。GitHub为了通知到Jenkins，需要使用公网IP访问Jenkins，这一点和VSTS是一样的，当然也可以采用反向代理服务。

16.5.1　配置GitHub

配置GitHub的步骤如下：

（1）在GitHub创建账号并登录后，单击账号头像，选择Settings→Developer Settings→Personal Access Tokens选项。然后如图16-43所示，单击Generate new token按钮生成访问令牌（token）。该令牌的功能与OAuth的access token类似，支持第三方应用访问GitHub API[①]。

Personal access tokens　　Generate new token

Need an API token for scripts or testing? Generate a personal access token for quick access to the GitHub API.

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.

图16-43　创建个人访问令牌

① GitHub. GitHub Developer Guide[OL]. [2016]. https://developer.github.com/.

（2）如图16-44所示，填写Token description页面并选择token的权限范围（scope）。本示例需要设置以下3种权限：

- repo：用来访问代码库。
- admin:repo_hook：操作hooks，包括读写、删除等操作。
- repo:status：操作变更状态。

Token description

tokenfortestdemo

What's this token for?

Select scopes

Scopes define the access for personal tokens. Read more about OAuth scopes.

☑ repo	Full control of private repositories
☑ repo:status	Access commit status
☑ repo_deployment	Access deployment status
☑ public_repo	Access public repositories
☐ admin:org	Full control of orgs and teams
☐ write:org	Read and write org and team membership
☐ read:org	Read org and team membership
☐ admin:public_key	Full control of user public keys
☐ write:public_key	Write user public keys
☐ read:public_key	Read user public keys
☑ admin:repo_hook	Full control of repository hooks
☑ write:repo_hook	Write repository hooks
☑ read:repo_hook	Read repository hooks

图16-4　选择权限范围

（3）单击Generate token按钮，即可获得创建好的token文本。

（4）创建好代码库后，选择Settings→Webhooks→Add webhook选项。在Payload URL输入框内输入Jenkins的监听端口http://[JenkinsAddress]:[JenkinsPort]/github-webhook/，其中JenkinsAddress和JenkinsPort需要是读者实际使用的Jenkins服务的地址和端口号。单击Add webhook按钮，创建webhook，如图16-45所示。当有新的代码变更时，GitHub可以通过配置好的webhook通知Jenkins。Webhook的默认事件类型是push，关于其他事件类型，读者可以参考网址https://developer.github.com/webhooks/中的资料。

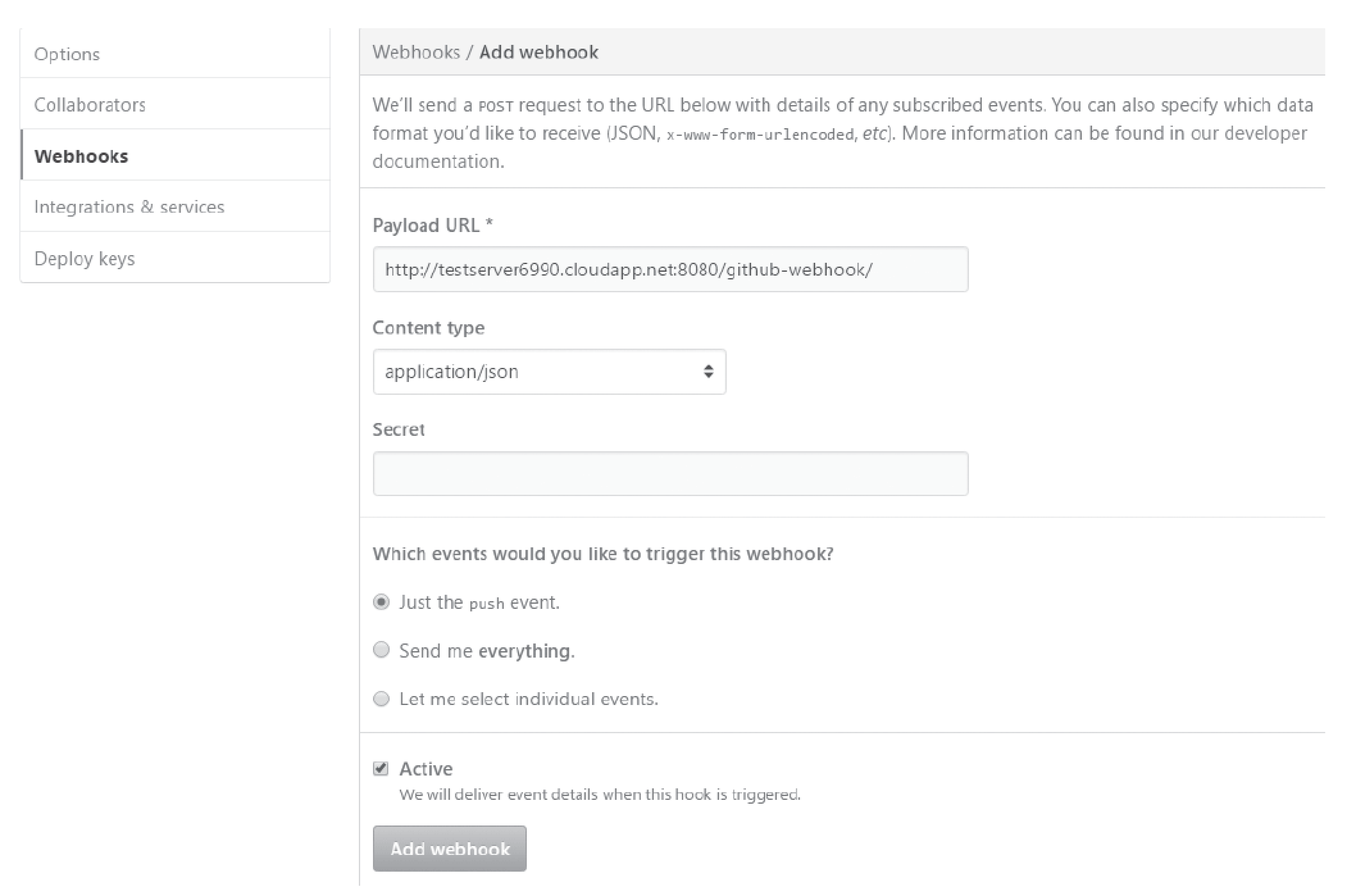


图16-45　创建Webhook

16.5.2　配置Jenkins

配置Jenkins的步骤如下：

（1）登录Jenkins并安装GitHub plugin，该插件默认在初次启动Jenkins的时候被安装。

（2）选择Manage Jenkins→Configure System→GitHub→Add GitHub Server。如图16-46所示，在API URL输入框内输入https://api.github.com，在Credentials选项中提供上一节创建的个人访问令牌。单击Add按钮，创建一个类型为Secret text的凭据，在Secret输入框内输入上一节获得的令牌文本，如图16-47所示。

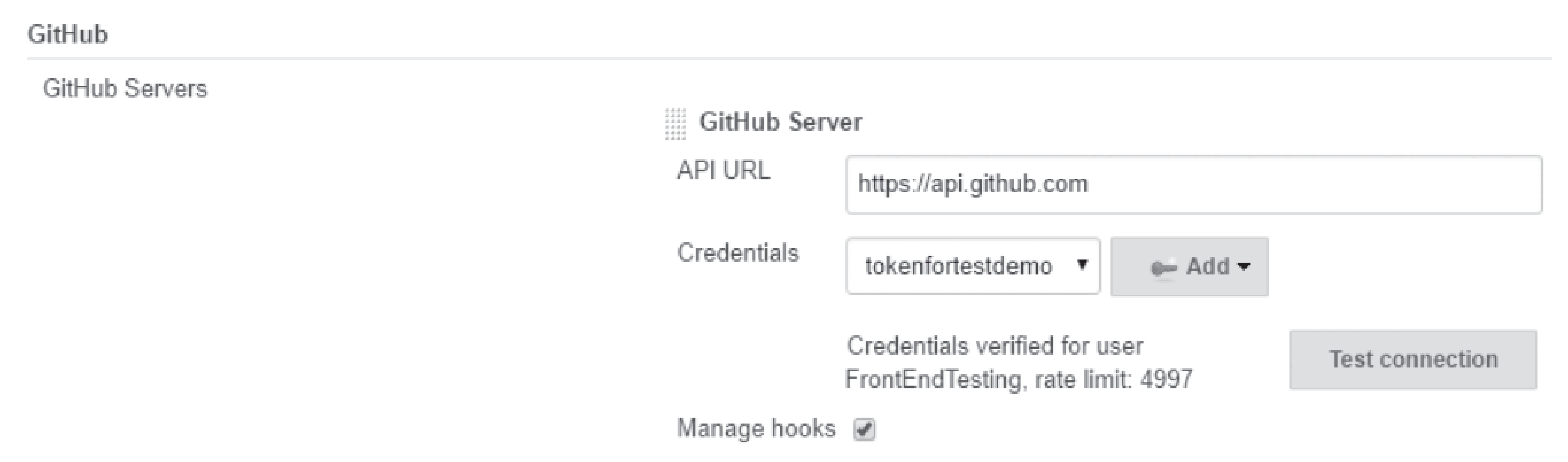


图16-46　配置GitHub Server

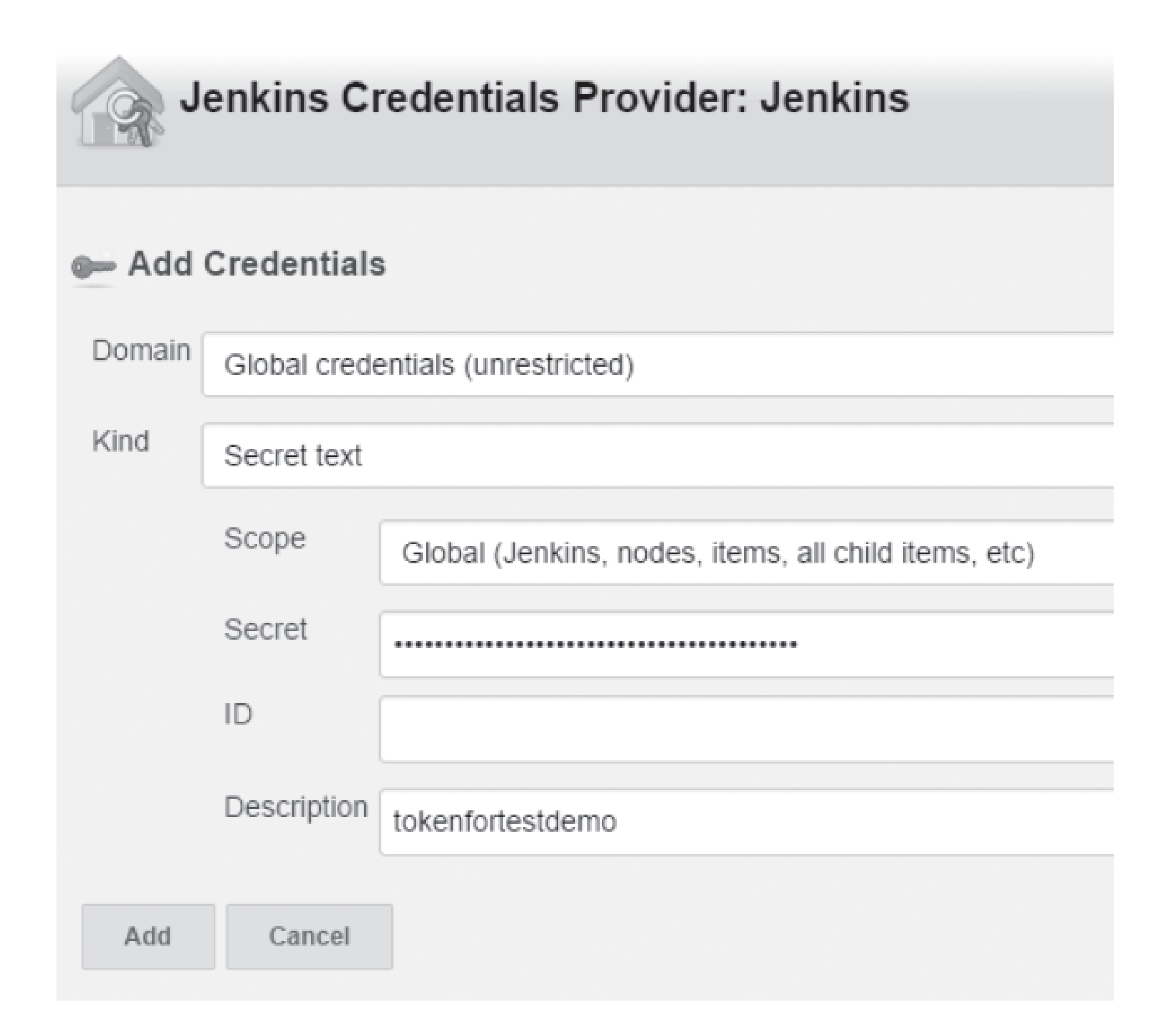


图16-47　设置令牌凭据

16.5.3　配置构建任务

配置构建任务的步骤如下：

（1）在Jenkins中创建新的构建任务，如图16-48所示。勾选GitHub project复选框并将Project url设置为GitHub中创建的代码库地址。

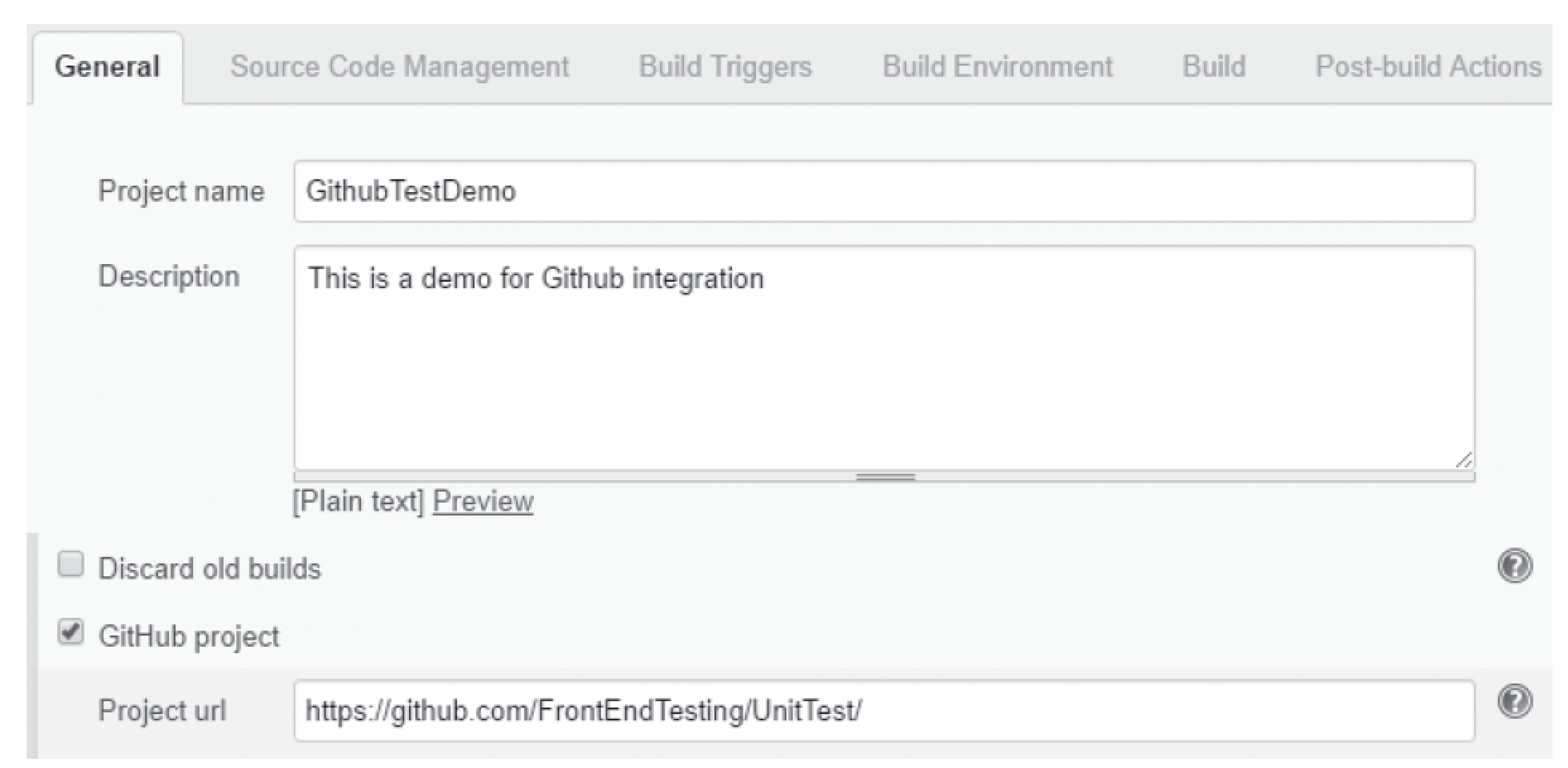


图16-48　新建构建任务

（2）在Source Code Management页面内，选择Git单选按钮，并设置GitHub代码库的git访问地址，如图16-49所示。

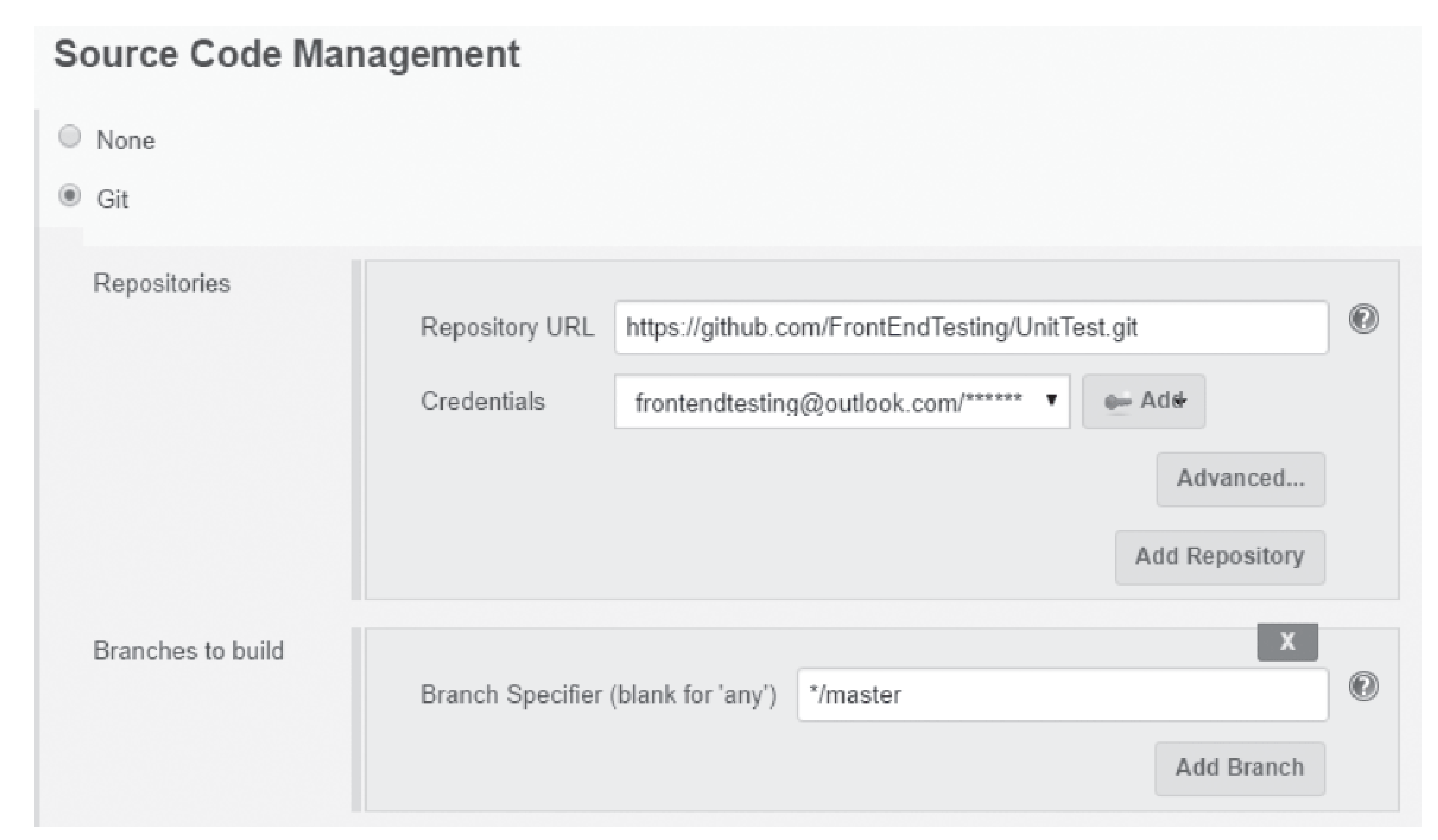


图16-49　设置Git源

完成以上步骤后，Jenkins就可以访问GitHub或者在代码变更时得到GitHub的通知。GitHub关于单元测试和自动化测试的集成步骤，与TFS和VSTS一致，这里不再赘述。

最后，请读者理解持续集成是一个不断演进，持续深化的过程。尽管在实践中可能使用到不同的CI系统，不同的构建脚本，但中心思想都是通过持续构建提升软件发布质量。衷心祝愿读者将已学的单元测试和自动化测试知识融入到自己的集成实践之中，不断改进测试效率，将软件质量提升到新高度。